STATEFLOW™

for Use with Simulink®

Modeling

Simulation

Implementation

User's Guide

Version 1

How to Contact The MathWorks:

508-647-7000 Phone

508-647-7001 Fax

The MathWorks, Inc.
24 Prime Park Way
Natick, MA 01760-1500 Mail

`http://www.mathworks.com` Web
`ftp.mathworks.com` Anonymous FTP server
`comp.soft-sys.matlab` Newsgroup

`support@mathworks.com` Technical support
`suggest@mathworks.com` Product enhancement suggestions
`bugs@mathworks.com` Bug reports
`doc@mathworks.com` Documentation error reports
`subscribe@mathworks.com` Subscribing user registration
`service@mathworks.com` Order status, license renewals, passcodes
`info@mathworks.com` Sales, pricing, and general information

Stateflow User's Guide

© COPYRIGHT 1997 - 1998 by The MathWorks, Inc. All Rights Reserved.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

U.S. GOVERNMENT: If Licensee is acquiring the Programs on behalf of any unit or agency of the U.S. Government, the following shall apply: (a) For units of the Department of Defense: the Government shall have only the rights specified in the license under which the commercial computer software or commercial software documentation was obtained, as set forth in subparagraph (a) of the Rights in Commercial Computer Software or Commercial Software Documentation Clause at DFARS 227.7202-3, therefore the rights set forth herein shall apply; and (b) For any other unit or agency: NOTICE: Notwithstanding any other lease or license agreement that may pertain to, or accompany the delivery of, the computer software and accompanying documentation, the rights of the Government regarding its use, reproduction, and disclosure are as set forth in Clause 52.227-19 (c)(2) of the FAR.

MATLAB, Simulink, Handle Graphics, and Real-Time Workshop are registered trademarks and Stateflow and Target Language Compiler are trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History: May 1997 First printing
January 1998 Revised for 5.2 (online version)

Contents

Before You Begin

Related Products and Documentation ..... **xxii**
Requirements ..... **xxii**

How to Use This Guide ..... **xxvi**
Chapter Quick Reference ..... **xxvi**
Typographical Conventions ..... **xxvii**

Installation ..... **xxviii**

1 Introduction

Overview **1-2**
What Is Stateflow? **1-2**
Examples of Stateflow Applications **1-3**
Stateflow Components **1-3**
Design Approaches **1-4**

Finite State Machine Concepts **1-6**
What Is a Finite State Machine? **1-6**
FSM Representations **1-6**
Stateflow Representations **1-6**
Notations **1-7**
Semantics **1-7**
References **1-7**

Anatomy of a Model and Machine **1-8**
The Simulink Model and Stateflow Machine **1-8**
Defining Stateflow Interfaces **1-10**
Stateflow Diagram Objects **1-11**

Using Stateflow **1-22**
Stateflow Typical Tasks **1-22**
Sample Solution **1-22**
Creating a Simulink Model **1-23**
Creating a Stateflow Diagram **1-26**
Defining the Stateflow Interface **1-30**
Running a Simulation **1-32**
Debugging **1-33**
Generating Code **1-35**

2 Using the Graphics Editor

The Graphics Editor Main Window **2-2**
Overview **2-2**
Main Window Components **2-2**
Drawing Area **2-3**
Message Area **2-3**
Zooming Options **2-4**
Scroll Bars **2-4**
Stateflow Diagram (Stateflow Block) Name **2-4**
Modified Flag **2-5**
Menu Bar **2-5**
Toolbar **2-6**
Productivity Boosts **2-7**
Virtual Scrollbar **2-7**

Shortcut Menus **2-10**
Overview **2-10**
Menu Options Available on All Shortcut Menus **2-10**
General Shortcut Menu **2-12**
State Shortcut Menu **2-13**
Transition Shortcut Menu **2-14**
Junction Shortcut Menu **2-14**

Using the Graphics Editor **2-15**
Object Button Modes **2-15**
Canceling Operations **2-15**
Selecting and Deselecting Objects **2-15**
Cutting and Pasting Objects **2-16**
Copying Objects **2-16**

Working with Charts (Stateflow Diagrams) **2-18**
Overview **2-18**
What Can You Do with Charts? **2-18**
Editing Chart (Stateflow Diagram) Properties **2-19**
Chart Hyperlinks Example **2-21**
Chart Update Methods **2-22**
Printing Large Stateflow Diagrams **2-23**

Working with States **2-24**
What Can You Do with States? **2-24**
Specifying Decomposition **2-24**
Creating/Deleting States **2-25**
Labeling States **2-26**
Grouping and Ungrouping States **2-26**
Showing and Hiding States **2-27**
Resizing States **2-27**
Changing Font Size **2-27**
Changing Arrowhead Size **2-28**
Moving States **2-28**
Editing State Properties **2-29**

Working with Transitions **2-31**
What Can You Do with Transitions? **2-31**
Creating/Deleting Transitions **2-31**
What Is a Default Transition? **2-32**
Creating Default Transitions **2-32**
Editing Transition Attach Points **2-32**
Labeling Transitions **2-33**
Valid Labels **2-34**
Changing Font Size **2-34**
Changing Arrowhead Size **2-34**
Moving Transition Labels **2-35**
Editing Transition Properties **2-36**

Working with Junctions **2-38**
What Can You Do with Junctions? **2-38**
Creating/Deleting Junctions **2-38**
Changing Size **2-39**
Changing Arrowhead Size **2-39**
Moving a Junction **2-40**
Editing Junction Properties **2-40**

Main Editor Menus **2-42**
Overview **2-42**
File Menu **2-43**
Print As Submenu **2-44**
Edit Menu **2-45**
Simulation Menu **2-45**
Style Menu **2-46**
Tools Menu **2-47**
Add Menu **2-48**
Help Menu **2-49**

Defaults File **2-50**
What Is the Defaults File? **2-50**
What Do You Need to Know about the Defaults File? **2-50**
Machine Defaults **2-50**
Defining Color **2-51**
Chart Defaults **2-51**
Target Defaults **2-52**
State Defaults **2-53**
Transition Defaults **2-53**
Junction Defaults **2-53**
Data Defaults **2-54**
Event Defaults **2-54**

3 Events and Data

Overview **3-2**
Events and Data **3-2**
Where to Find More Information on Creating, Modifying, and Deleting Events and Data **3-4**

Events **3-5**
What Is an Event? **3-5**
Event Properties **3-5**
How Are Events Broadcast? **3-6**
Getting Started with Events **3-6**
Event Name **3-6**
Event Description **3-7**
Event Parent Types **3-7**
Events, Hierarchy, and Scope **3-9**
Event Scope **3-10**
Table of Event Parent and Scope Combinations **3-15**
Event Ports **3-16**
Event Indices **3-16**
Event Triggers **3-17**
Determining Overall Input Trigger Type **3-18**
Combining Function Call and Edge Triggers **3-19**

Implicit Local Events **3-20**
Overview **3-20**
Types of Implicit Events **3-20**
Example: Implicit Local Event **3-21**

Data **3-22**
What Is a Data Object? **3-22**
Getting Started with Data **3-23**
Data Name **3-24**
Data Description **3-24**
Data Parent Types **3-25**
Data, Hierarchy, and Scope **3-26**
Data Scope **3-27**
Table of Data Scope and Parent Combinations **3-32**
Data Ports **3-32**
Data Attributes **3-33**

4 Defining Stateflow Interfaces

Overview **4-2**
Interfaces to Stateflow **4-2**
Typical Tasks to Define Stateflow Interfaces **4-2**
Where to Find More Information on Events and Data **4-3**

Defining the Stateflow Block Update Method **4-4**
Stateflow Block Update Methods **4-4**
Defining a Triggered Stateflow Block **4-5**
Example: Triggered Stateflow Block **4-5**
Defining a Sampled Stateflow Block **4-5**
Example: Sampled Stateflow Block **4-6**
Defining an Inherited Stateflow Block **4-6**
Example: Inherited Stateflow Block **4-7**
Defining a Continuous Stateflow Block **4-7**
Considerations in Choosing Continuous Update **4-7**
Example: Continuous Stateflow Block **4-8**

Defining Output to Simulink Event Triggers **4-10**
Overview **4-10**
Defining Function Call Output Events **4-10**
Example: Using Function Call Output Events **4-11**
Example: Function Call Semantics **4-12**
Defining Edge-Triggered Output Events **4-13**
Example: Edge-Triggered Semantics **4-13**

Event Inputs from Simulink **4-16**
Tasks to Define an Event Input from Simulink **4-16**
Add an Event Choosing a Chart as the Parent **4-16**
Choose Input from Simulink as the Scope **4-16**
Select the Trigger **4-17**
Apply the Changes **4-17**

Data Inputs from Simulink **4-18**
Tasks to Define a Data Input from Simulink **4-18**
Add a Data Object Choosing a Chart as the Parent **4-18**
Choose Input from Simulink as the Scope **4-18**
Specify Data Attributes **4-19**
Apply and Save the Changes **4-19**

Event Outputs to Simulink **4-20**
Tasks to Define an Event Output to Simulink **4-20**
Add an Event Choosing a Chart as the Parent **4-20**
Choose Output to Simulink as the Scope **4-20**
Apply the Changes **4-21**

Data Outputs to Simulink **4-22**
Tasks to Define a Data Output to Simulink **4-22**
Add a Data Object Choosing a Chart as the Parent **4-22**
Choose Output to Simulink as the Scope **4-22**
Specify Data Attributes **4-23**
Apply the Changes **4-23**

MATLAB Workspace **4-24**
What Is the MATLAB Workspace? **4-24**
Using the MATLAB Workspace **4-24**

Defining the Interface to External Sources **4-25**
What Are External Sources? **4-25**
Exported Events **4-25**
Imported Events **4-27**
Exported Data **4-28**
Imported Data **4-29**

5 Exploring and Searching

Overview **5-2**
Exploring and Searching **5-2**
Where to Find More Information on Events and Data **5-3**
Typical Exploring Tasks **5-3**
Typical Searching Tasks **5-4**

Stateflow Explorer User Interface **5-5**
Explorer Main Window **5-5**
Moving Objects/Changing Parent **5-8**
Moving Objects/Changing Index and Port Order **5-8**
Deleting Objects **5-8**
Event Properties Dialog Box **5-9**
Data Properties Dialog Box **5-12**

Example: Getting Started Adding Events and Data **5-16**
Task Overview **5-16**
Solution Model and Stateflow Diagram **5-16**
What Are the Events? **5-17**
Add Local Events **5-18**
What Is the Data? **5-18**
Adding Local Data **5-19**

Stateflow Finder User Interface **5-20**
Finder Dialog Box **5-20**
Finder Display Area **5-23**

Searching for a Wildcard String **5-26**
Example **5-26**

Searching for an Exact String Match **5-28**
Example **5-28**

Searching for All Objects of a Specified Object Type **5-30**
Example **5-30**

Searching Based on Multiple Conditions **5-32**
Example **5-32**

Searching for an Object to Edit Properties **5-34**
Example **5-34**

6 Notations

Overview **6-2**
What Is Meant by Notation? **6-2**
Motivation Behind the Notation **6-2**
How Is the Notation Checked? **6-3**
What Are the Graphical Objects? **6-3**
What Are the Nongraphical Objects? **6-4**
What Is the Data Dictionary? **6-4**
Keywords **6-5**
How Is Hierarchy Represented? **6-6**
Example: Representing State Hierarchy **6-7**
Example: Representing Transition Hierarchy **6-8**
Example: Representing Event Hierarchy **6-9**

States **6-10**
Overview **6-10**
What Is a State Decomposition? **6-10**
Active and Inactive States **6-11**
What Are Combination States? **6-12**
Labeling a State **6-13**
Example: Labeling a State **6-14**
Example: Unique State Names **6-15**

Transitions **6-17**
What Is a Transition? **6-17**
Labeling a Transition **6-18**
Example: Transition Label **6-18**
What Is a Valid Transition? **6-19**
Types of Transitions **6-20**
Example: Transitions to and from Exclusive (OR) States **6-21**
Example: Transitions to and from Junctions **6-21**
Example: Transitions to and from Exclusive OR Superstates **6-22**
Example: Transitions to and from Substates **6-23**
What is a Default Transition? **6-24**
Labeling of Default Transitions **6-24**
Example: Use of Default Transitions **6-25**
Example: Default Transition to a Junction **6-26**
Example: Default Transition with a Label **6-26**
What Is an Inner Transition? **6-27**
Example One: Before Using an Inner Transition **6-27**
Example One: Inner Transition to a Connective Junction . . . **6-28**
Example Two: Before Using an Inner Transition **6-29**
Example Two: Inner Transition to a Connective Junction . . . **6-29**
Example: Inner Transition to a History Junction **6-30**
What Is a Self Loop Transition? **6-31**

Connective Junctions **6-32**
What Is a Connective Junction? **6-32**
What Is Flow Diagram Notation? **6-32**
Example: Connective Junction with All Conditions Specified **6-33**
Example: Connective Junction with One Unconditional Transition **6-33**
Example: Connective Junction Special Case - Self Loop **6-34**
Example: Connective Junction and For Loops **6-35**
Example: Flow Diagram Notation **6-36**
Example: Connective Junction from a Common Source to Multiple Destinations **6-37**
Example: Connective Junction Common Destination **6-38**
Example: Connective Junction Common Events **6-38**

History **6-40**
What Is a History Junction? **6-40**
Example: Use of History Junctions **6-40**
History Junctions and Inner Transitions **6-41**

Action Language **6-42**
What Is an Action Language? **6-42**
Objects with Actions **6-42**
Transition Action Notation **6-43**
State Action Notation **6-43**
Action Language Components **6-44**
Bit Operations **6-44**
Binary Operations **6-45**
Unary Operations **6-47**
Unary Actions **6-48**
Assignment Operations **6-48**
User-Written Functions **6-49**
ml() Functions **6-50**
ml. Name Space Operator **6-53**
Choosing Between the ml() function and the ml Name Space Operator **6-55**
Data and Event Arguments **6-56**
Arrays **6-56**
Event Broadcasting **6-56**
Directed Event Broadcasting **6-58**
Example: Directed Event Broadcast Using send **6-59**

Example: Directed Event Broadcast Using Qualified Event Names 6-60
Conditions 6-61
Time Symbol 6-61
Literals 6-62
Continuation Symbols 6-62
Comments 6-62
Use of the Semicolon 6-62

7 Semantics

Overview 7-2
What Are Semantics? 7-2
List of Semantic Examples 7-2

Event-Driven Effects on Semantics 7-5
What Does Event-Driven Mean? 7-5
Top-Down Processing of Events 7-5
Semantics of Active and Inactive States 7-5
Semantics of State Actions 7-7
Semantics of Transitions 7-7

Transitions to and from Exclusive (OR) States 7-8
Example: Processing of One Event 7-8
Example: Processing of a Second Event 7-9
Example: Processing of a Third Event 7-10
Example: Transition from a Substate to a Substate 7-11

Condition Actions **7-13**
Example: Actions Specified as Condition Actions **7-13**
Example: Actions Specified as Condition and Transition Actions **7-14**
Example: Using Condition Actions in For Loop Construct ... **7-15**
Example: Using Condition Actions to Broadcast Events to Parallel (AND) States **7-16**
Example: Cyclic Behavior to Avoid When Using Condition Actions **7-17**

Default Transitions **7-18**
Example: Default Transition in an Exclusive (OR) Decomposition **7-18**
Example: Default Transition to a Junction **7-20**
Example: Default Transition and a History Junction **7-21**
Example: Labeled Default Transitions **7-23**

Inner Transitions **7-25**
Example: Processing One Event Within an Exclusive (OR) State **7-25**
Example: Processing a Second Event Within an Exclusive (OR) State **7-26**
Example: Processing a Third Event Within an Exclusive (OR) State **7-27**
Example: Processing One Event with an Inner Transition to a Connective Junction **7-28**
Example: Processing a Second Event with an Inner Transition to a Connective Junction **7-29**
Example: Inner Transition to a History Junction **7-31**

Connective Junctions **7-33**
Example: If-Then-Else Decision Construct **7-33**
Example: Self Loop **7-34**
Example: For Loop Construct **7-35**
Example: Flow Diagram Notation **7-37**
Example: Transitions from a Common Source to Multiple Destinations **7-39**
Example: Transitions from Multiple Sources to a Common Destination **7-40**
Example: Transitions from a Source to a Destination Based on a Common Event **7-41**

Event Actions **7-43**
Example: Event Actions and Superstates **7-43**

Parallel (AND) States **7-45**
Example: Event Broadcast State Action **7-45**
Example: Event Broadcast Transition Action (Nested Event Broadcast) **7-50**
Example: Event Broadcast Condition Action **7-54**

Directed Event Broadcasting **7-58**
Example: Directed Event Broadcast Using send **7-58**
Example: Directed Event Broadcasting Using Qualified Event Names **7-60**

Execution Order **7-62**
Overview **7-62**
Execution Order Guidelines **7-62**
Parallel (AND) States **7-65**

Semantic Rules Summary **7-67**
Event Broadcasting Rules **7-67**
Rules for Processing States **7-68**
Rules for Processing Transitions **7-69**
Rules When Taking Transition Paths **7-70**

8 Tutorial: Entering a Stateflow Diagram

Overview **8-2**
- Objectives **8-2**
- Provided Models for This Chapter **8-2**
- Carousel Tutorial Example **8-2**
- Basic Operation **8-2**

Designing the Carousel Stateflow Diagram **8-4**
- Overview **8-4**
- The Design Questions **8-4**
- What Are the States? **8-4**
- What Is a State Decomposition? **8-5**
- What Are the State Labels? **8-6**
- Will Junctions Simplify the Design? **8-6**
- What Are the Transitions? **8-6**
- Are There Default Transitions? **8-7**
- Are There Events? **8-7**
- Is There Data? **8-7**

Procedure **8-8**
- Overview **8-8**
- Create a Simulink Model **8-9**
- Sample Stateflow Diagram **8-10**
- Create and Label the OR States **8-10**
- Create and Label the Superstates **8-13**
- Create and Label the AND States **8-14**
- Create the Transitions **8-15**
- Label the Transitions **8-16**
- Create Default Transitions **8-17**
- Add and Define Event Objects **8-18**
- Add and Define Data Objects **8-20**
- Save the Model **8-23**

Run the Simulation for the Provided Solution **8-24**
Overview **8-24**
Carousel Simulink Model Solution **8-24**
Carousel Operator's Console **8-24**
Carousel Solution Stateflow Diagram **8-25**
Procedure **8-25**

Run the Carousel Demo **8-27**
Overview **8-27**
Carousel Operator's Console **8-27**

9 Generating Code

Overview **9-2**
Overview **9-2**
Stateflow Coder **9-2**
Stateflow Dynamic Checker **9-3**
Compile/Build Options **9-3**
Typical Tasks to Generate Code **9-4**
Typical Code Generation Sequence **9-5**
Coder File and Coder Function **9-5**

Using the Notation Efficiently **9-7**
Where to Find Information on Creating a Stateflow Diagram . **9-7**
Designing for Efficiency **9-7**
Optimizing for Speed **9-7**
Optimizing for Code Size **9-8**
Optimizing for Memory Use **9-9**

Parsing **9-11**
Parser **9-11**
Parse the Machine or the Stateflow Diagram **9-11**

Adding and Specifying a Stand-Alone Target **9-16**
Stand-Alone Targets **9-16**
Invoke the Explorer **9-16**
Add a Stand-Alone Target **9-17**

Customizing the Target **9-20**
Overview **9-20**
Potential Customizations **9-20**
Displaying the Target Configuration sfun Dialog Box **9-21**
Displaying the Target Configuration rtw Dialog Box **9-22**
Displaying the Target Configuration Dialog Box **9-23**
Coder Command Customization **9-23**
Make Command Customization **9-23**
Enable Debugging/Animation **9-24**
Enable Dynamic Checker Options **9-24**
Add Customized Code **9-25**
Specify a Document Link **9-25**

Compiling Code/Building Targets **9-26**
Compile/Build Options **9-26**
Compiling/Building a Simulation Target **9-27**
Compiling/Building Targets with RTW **9-27**
Compiling/Building Stand-Alone Targets **9-28**

Typical Code Generation Related Error Messages **9-30**
Overview **9-30**
Typical Parsing Error Messages **9-30**
Code Generation Error Messages **9-32**
Compilation Error Messages **9-32**

10 Debugging

Overview **10-2**
- Debugging **10-2**
- Stateflow Dynamic Checker **10-2**
- Typical Debugging Tasks **10-3**
- Including Debugging in the Target Build **10-3**
- Breakpoints **10-3**
- Runtime Debugging **10-4**

Stateflow Debugger User Interface **10-6**
- Debugger Main Window **10-6**
- Status Display Area **10-8**
- Break On Controls **10-8**
- Debugger Action Control Buttons **10-9**
- Dynamic Checker Options **10-10**
- Animation Controls **10-10**
- Display Controls **10-11**
- MATLAB Command Field **10-11**

Debugging Runtime Errors **10-12**
- Example Stateflow Diagram **10-12**
- Typical Scenario to Debug Runtime Errors **10-13**
- Create the Model and Stateflow Diagram **10-13**
- Define the sfun Target **10-14**
- Invoke the Debugger and Choose Debugging Options **10-14**
- Start the Simulation **10-14**
- Debug the Simulation Execution **10-14**
- Resolve Runtime Error and Repeat **10-15**
- Solution Stateflow Diagram **10-15**

Debugging State Inconsistencies **10-16**
- What is a State Inconsistency? **10-16**
- Causes of State Inconsistency **10-16**
- Detecting State Inconsistency **10-16**
- Example: State Inconsistency **10-17**

Debugging Conflicting Transitions **10-18**
- What is a Conflicting Transition? **10-18**

Detecting Conflicting Transitions **10-18**
Example: Conflicting Transition **10-18**

Debugging Data Range Violations **10-20**
What is a Data Range Violation? **10-20**
Detecting Data Range Violations **10-20**
Example: Data Range Violation **10-20**

Debugging Cyclic Behavior **10-21**
What is Cyclic Behavior? **10-21**
Detecting Cyclic Behavior **10-21**
Example: Cyclic Behavior **10-22**
Example: Flow Cyclic Behavior Not Detected **10-23**
Example: Noncyclic Behavior Flagged as a Cycle **10-24**

11 Reference

sfnew 11-3
sfexit 11-4
sfsave 11-5
stateflow 11-6
sfprint 11-9
sfhelp 11-10

A Glossary

Before You Begin

Related Products and Documentation xxii
Requirements xxii

How to Use This Guide xxvi
Chapter Quick Reference xxvi
Typographical Conventions xxvii

Installation xxviii

Related Products and Documentation

Requirements

Stateflow™ is a multiplatform product, running on Windows95, Windows NT, and UNIX systems.

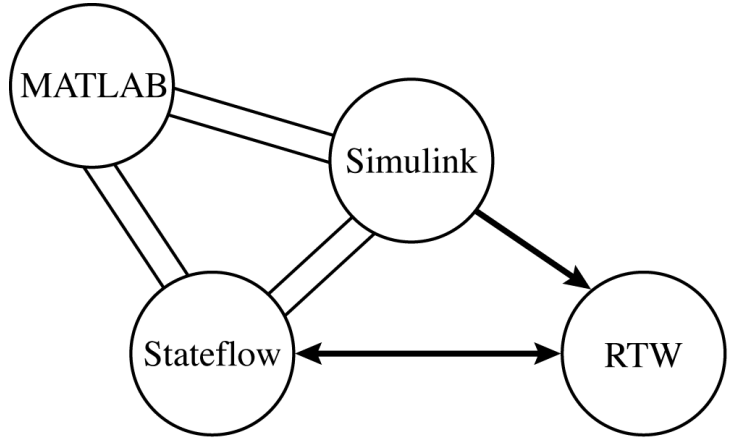
Stateflow requires:

- MATLAB® 5.2
- Simulink® 2.2
- A C or C++ compiler for creating MATLAB MEX-files on your platform.For a list of all the compilers supported by MATLAB, see the MathWorks Technical Support Department's Technical Notes at:

  `http://www.mathworks.com/support/tech-notes/#mex`

See the *MATLAB Application Program Interface Guide* for information on how to ensure proper installation for integration into the MATLAB environment.

In addition, you can optionally use the Real-Time Workshop® to generate floating point code for Simulink portions of the model.



What Is MATLAB?

MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. Typical uses include:

- Math and computation
- Algorithm development
- Modeling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including graphical user interface building

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This allows you to solve many technical computing problems, especially those with matrix and vector formulations, in a fraction of the time it would take to write a program in a scalar noninteractive language such as C or Fortran.

The name MATLAB stands for *matrix laboratory*. MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects, which together represent the state-of-the-art in software for matrix computation.

MATLAB has evolved over a period of years with input from many users. In university environments, it is the standard instructional tool for introductory and advanced courses in mathematics, engineering, and science. In industry, MATLAB is the tool of choice for high-productivity research, development, and analysis.

MATLAB features a family of application-specific solutions called *toolboxes*. Very important to most users of MATLAB, toolboxes allow you to *learn* and *apply* specialized technology. Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation, and many others.

See the MATLAB documentation set for more information.

What Is Simulink?

Simulink is a software package for modeling, simulating, and analyzing dynamic systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate, i.e., have different parts that are sampled or updated at different rates.

For modeling, Simulink provides a graphical user interface (GUI) for building models as block diagrams, using click-and-drag mouse operations. With this interface, you can draw the models just as you would with pencil and paper (or as most textbooks depict them). This is a far cry from previous simulation packages that require you to formulate differential equations and difference equations in a language or program. Simulink includes a comprehensive block library of sinks, sources, linear and nonlinear components, and connectors. You can also customize and create your own blocks.

Models are hierarchical, so you can build models using both top-down and bottom-up approaches. You can view the system at a high-level, then double-click on blocks to go down through the levels to see increasing levels of model detail. This approach provides insight into how a model is organized and how its parts interact.

After you define a model, you can simulate it, using a choice of integration methods, either from the Simulink menus or by entering commands in MATLAB's command window. The menus are particularly convenient for interactive work, while the command-line approach is very useful for running a batch of simulations (for example, if you are doing Monte Carlo simulations or want to sweep a parameter across a range of values). Using scopes and other display blocks, you can see the simulation results while the simulation is running. In addition, you can change parameters and immediately see what happens, for "what if" exploration. The simulation results can be put in the MATLAB workspace for postprocessing and visualization.

Model analysis tools include linearization and trimming tools, which can be accessed from the MATLAB command line, plus the many tools in MATLAB and its application toolboxes. And because MATLAB and Simulink are integrated, you can simulate, analyze, and revise your models in either environment at any point.

Stateflow is part of this environment. The Stateflow block is a masked Simulink model. Stateflow builds an S-function that corresponds to each Stateflow machine. This S-function is the agent Simulink interacts with for simulation and analysis.

The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow diagrams into Simulink models, you can add event-driven behavior to Simulink simulations. You create models that represent both data and control flow by combining Stateflow blocks with the standard Simulink blocksets. These combined models are simulated using Simulink.

The *Using Simulink* document describes how to work with Simulink. It explains how to manipulate Simulink blocks, access block parameters, and connect blocks to build models. It also provides reference descriptions of each block in the standard Simulink libraries.

What Is Real-Time Workshop?

Real-Time Workshop (RTW) is an automatic C language code generator for Simulink. It produces C code directly from Simulink block diagram models and automatically builds programs that can be run in real-time in a variety of environments.

Stateflow works in conjunction with Real-Time Workshop in these ways:

- Stateflow generates high quality integer-based code (runs on a fixed-point processor; a floating-point processor is not required).
- RTW generates floating-point-based code (a floating-point processor is required).
- Stateflow blocks are fully integrated for real-time code generation and compatible with RTW.

Note: If you want to target Stateflow and RTW generated code for the same processor, you need a floating-point processor.

How to Use This Guide

Chapter Quick Reference

If you are new to the Stateflow environment, go to Chapter 1, “Introduction” to get an overview and a quick start.

If you want to get started right away, go to Chapter 8, “Tutorial: Entering a Stateflow Diagram” for step by step instruction on creating a Stateflow diagram.

For information on using the graphics editor refer to Chapter 2, “Using the Graphics Editor.”

Chapter 3, “Events and Data” describes the nongraphical objects that are essential to completing and defining interfaces to the Stateflow diagram.

Chapter 4, “Defining Stateflow Interfaces” describes the types of interfaces and how to define them.

For information on using the Stateflow Explorer and the Stateflow Finder, see Chapter 5, “Exploring and Searching.”

Use Chapter 6, “Notations,” Chapter 7, “Semantics,” and Chapter 9, “Generating Code” together to understand the language used to communicate Stateflow diagram design information, how that notation is interpreted and implemented behind the scenes, and how to generate code.

See Chapter 10, “Debugging,” for information on debugging your simulation.

See Chapter 11, “Reference,” for information on specific functions and their syntax.

See the “Glossary” for definitions of key terms and concepts.

Typographical Conventions

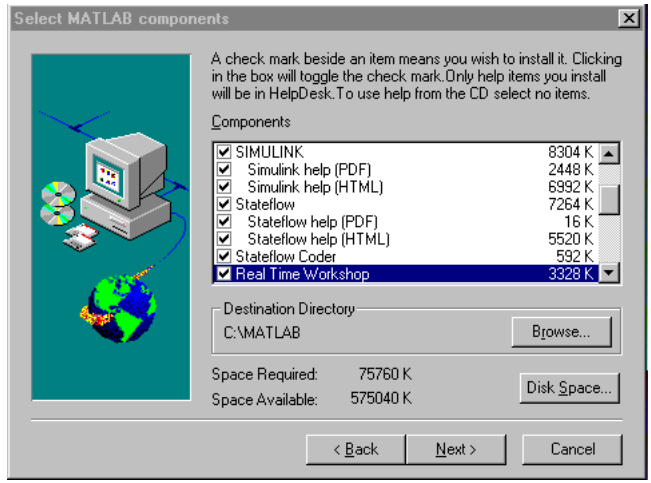
To Indicate...	This Guide Uses...	Example
New terms	*Italics*	An *array* is an ordered collection of information.
Keys, menu names, items, and GUI controls	**Boldface** with an initial capital letter	Choose the **File** menu.
Function names, variables, example code, user-entered text	`Monospace` type	The `sfopen` command opens a model.

Installation

Your platform specific MATLAB *Installation Guide* provides essentially all of the information you need to install Stateflow.

Prior to installing Stateflow, you must obtain a License File or Personal License Password from The MathWorks. The License File or Personal License Password identifies the products you are permitted to install and use.

As the installation process proceeds, a screen similar to this, where you can indicate which products to install, is displayed:


Select MATLAB components
A check mark beside an item means you wish to install it. Clicking in the box will toggle the check mark. Only help items you install will be in HelpDesk. To use help from the CD select no items.
Components
SIMULINK 8304 K
Simulink help (PDF) 2448 K
Simulink help (HTML) 6992 K
Stateflow 7264 K
Stateflow help (PDF) 16 K
Stateflow help (HTML) 5520 K
Stateflow Coder 592 K
Real Time Workshop 3328 K
Destination Directory
C:\MATLAB
Browse...
Space Required: 75760 K
Space Available: 575040 K
Disk Space...
< Back
Next >
Cancel

Stateflow and Stateflow Coder have certain product prerequisites that must be met for proper installation and execution:

Licensed Product	Prerequisite Products	Additional Information
Simulink 2.2	MATLAB 5.2	Allows installation of Simulink and Stateflow in Demo mode.
Stateflow	Simulink 2.2	Requires a C or C++ compiler for creating MATLAB MEX-files on your platform (see page xxii).
Stateflow Coder	Stateflow	Same as Stateflow.

If you experience installation difficulties and have Web access, connect to The MathWorks home page (`http://www.mathworks.com`). Look for the license manager and installation information under the `Tech Notes/FAQ` link under `Tech Support Info`.

1

Introduction

Overview 1-2
What Is Stateflow? 1-2
Examples of Stateflow Applications 1-3
Stateflow Components 1-3
Design Approaches 1-4

Finite State Machine Concepts 1-6
What Is a Finite State Machine? 1-6
FSM Representations 1-6
Stateflow Representations 1-6
Notations 1-7
Semantics 1-7
References 1-7

Anatomy of a Model and Machine 1-8
The Simulink Model and Stateflow Machine 1-8
Defining Stateflow Interfaces 1-10
Stateflow Diagram Objects 1-11

Using Stateflow 1-22
Stateflow Typical Tasks 1-22
Sample Solution 1-22
Creating a Simulink Model 1-23
Creating a Stateflow Diagram 1-26
Defining the Stateflow Interface 1-30
Running a Simulation 1-32
Debugging 1-33
Generating Code 1-35

Overview

What Is Stateflow?

Stateflow is a powerful graphical design and development tool for complex control and supervisory logic problems. Stateflow supports flow diagram notation as well as state transition notation. Using Stateflow you can:

- Visually model and simulate complex reactive systems based on *finite state machine* theory.
- Design and develop deterministic, supervisory control systems.
- Use flow diagram notation and state transition notation seamlessly in the same Stateflow diagram.
- Easily modify your design, evaluate the results, and verify the system's behavior at any stage of your design.
- Automatically generate integer or floating-point code directly from your design.
- Take advantage of the integration with the MATLAB and Simulink environments to model, simulate, and analyze your system.

Flow diagram notation is essentially logic represented without the use of states. In some cases, using flow diagram notation is a closer representation of the system's logic and avoids the use of unnecessary states. Flow diagram notation is an effective way to represent common code structures like `for` loops and `if-then-else` constructs.

Stateflow also provides clear, concise descriptions of complex system behavior using finite state machine theory, flow diagram notations, and state-transition diagrams. Stateflow brings system specification and design closer together. It is easy to create designs, consider various scenarios, and iterate until the Stateflow diagram models the desired behavior.

Examples of Stateflow Applications

A few of the types of applications that benefit from using the capabilities of Stateflow are:

- Embedded systems
  - Avionics (planes)
  - Automotive (cars)
  - Telecommunications (e.g., routing algorithms)
  - Commercial (computer peripherals, appliances, etc.)
  - Programmable logic controllers (PLCs) (process control)
  - Industrial (machinery)
- Man-machine interface (MMI)
  - Graphical user interface (GUI)
- Hybrid systems
  - Air traffic control systems (Digital Signal Processing (DSP) + Control + MMI)

Stateflow Components

Stateflow consists of these primary components:

- Stateflow graphics editor (see Chapter 2, “Using the Graphics Editor”)
- Stateflow Explorer (see Chapter 5, “Exploring and Searching”)
- Stateflow Finder (see Chapter 5, “Exploring and Searching”)
- Stateflow simulation code generator (see Chapter 9, “Generating Code”)
- Stateflow Debugger (see Chapter 10, “Debugging”)

Stateflow Coder is a separately available product and generates code for nonsimulation targets. (See Chapter 9, “Generating Code” for information relevant to Stateflow Coder.)

Stateflow Dynamic Checker supports runtime checking for conditions such as cyclic behavior and data range violations. The Dynamic Checker is currently available if you have a Stateflow license.

Design Approaches

Stateflow is used together with Simulink and optionally with the Real-Time Workshop (RTW), all running on top of MATLAB. MATLAB provides access to data, high-level programming, and visualization tools. The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink. Simulink supports development of continuous-time and discrete-time dynamic systems in a graphical block diagram environment. Stateflow diagrams are incorporated into Simulink models to enhance the new event-driven capabilities in Simulink (such as conditionally executed subsystems and event detection).

You can design a model starting with a Stateflow (control) perspective and then later build the Simulink model. You can also design a model starting from a Simulink (algorithmic) perspective and then later add Stateflow diagrams. You may have an existing Simulink model that would benefit by replacing Simulink logic blocks with Stateflow diagrams. The approach you use determines how, and in what sequence, you develop various parts of the model.

The collection of all Stateflow blocks in the Simulink model is a machine. When using Simulink together with Stateflow for simulation, Stateflow generates an S-function (MEX-file) for each Stateflow machine to support model simulation. This generated code is a simulation target and is called the `sfun` target within Stateflow.

Stateflow Coder generates integer or floating-point code based on the Stateflow machine. RTW generates code from the Simulink portion of the model and provides a framework for running generated Stateflow code in real-time. The code generated by Stateflow Coder is seamlessly incorporated into code generated by RTW. You may want to design a solution that targets code generated from both products for a specific platform. This generated code is specifically a RTW target and within Stateflow is called the `rtw` target.

Using Stateflow and Stateflow Coder you can generate code exclusively for the Stateflow machine portion of the Simulink model. This generated code is for stand-alone (nonsimulation) targets. You uniquely name this target within Stateflow.

In summary, the primary design approach options are:

- Use Stateflow together with Simulink for simulation.
- Use Stateflow, Stateflow Coder, Simulink, and RTW to generate target code for the complete model.
- Use Stateflow and Stateflow Coder to generate target code for a machine.

Finite State Machine Concepts

What Is a Finite State Machine?

A *finite state machine* (FSM) is a representation of an event-driven (reactive) system. In an event-driven system, the system transitions from one state (mode) to another prescribed state, provided that the condition defining the change is true.

For example, you can represent an automatic transmission in a car by a finite state machine. The transmission has a number of different modes of operation. The system transitions from being in neutral to being engaged in a gear when the driver shifts the stick from the neutral to engaged position.

FSM Representations

Traditionally, the relationships between the inputs, outputs, and states of an FSM are represented in a truth table. The resulting table describes the logic necessary to control the behavior of the system under study. Another approach to designing event-driven systems is to model the behavior of the system by describing it in terms of states and transitions. The state that is active is determined based on the occurrence of events under certain conditions. State-transition diagrams (STDs) and bubble diagrams are graphical representations based on this approach.

Stateflow Representations

Stateflow uses a variant of the finite state machine notation established by Harel [1].Using Stateflow, you create Stateflow diagrams. A Stateflow diagram is a graphical representation of a finite state machine where *states* and *transitions* form the basic building blocks of the system. You can also represent flow (stateless) diagrams using Stateflow. Stateflow provides a block that you include in a Simulink model. The collection of Stateflow blocks in a Simulink model is the Stateflow machine.

Additionally, Stateflow enables the representation of hierarchy, parallelism, and history. Hierarchy enables you to organize complex systems by defining a parent/offspring object structure. For example, you can organize states within other higher-level states. A system with parallelism can have two or more orthogonal states active at the same time. History provides the means to specify the destination state of a transition based on historical information.

These characteristics enhance the usefulness of this approach and go beyond what STDs and bubble diagrams provide.

Notations

A notation defines a set of objects and the rules that govern the relationships between those objects. Stateflow notation provides a common language to communicate the design information conveyed by a Stateflow diagram.

Stateflow notation consists of:

- A set of graphical objects
- A set of nongraphical text-based objects
- Defined relationships between those objects

See Chapter 6, “Notations,” for detailed information on Stateflow notations.

Semantics

Semantics describe how the notation is interpreted and implemented. A completed Stateflow diagram illustrates how the system will behave. A Stateflow diagram contains actions associated with transitions and states. The semantics describe in what sequence these actions take place during Stateflow diagram execution.

Knowledge of the semantics is important to make sound Stateflow diagram design decisions for code generation. Different use of notations results in different ordering of simulation and generated code execution.

The default semantics provided with the product are described in Chapter 7, “Semantics.”

References

For more information on finite state machine theory, consult these sources:

[1] Harel, David, “Statecharts: A Visual Formalism for Complex Systems,” *Science of Computer Programming 8*, 1987, pages 231-274.

[2] Hatley, Derek J. and Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House Publishing Co., Inc., NY, 1988.

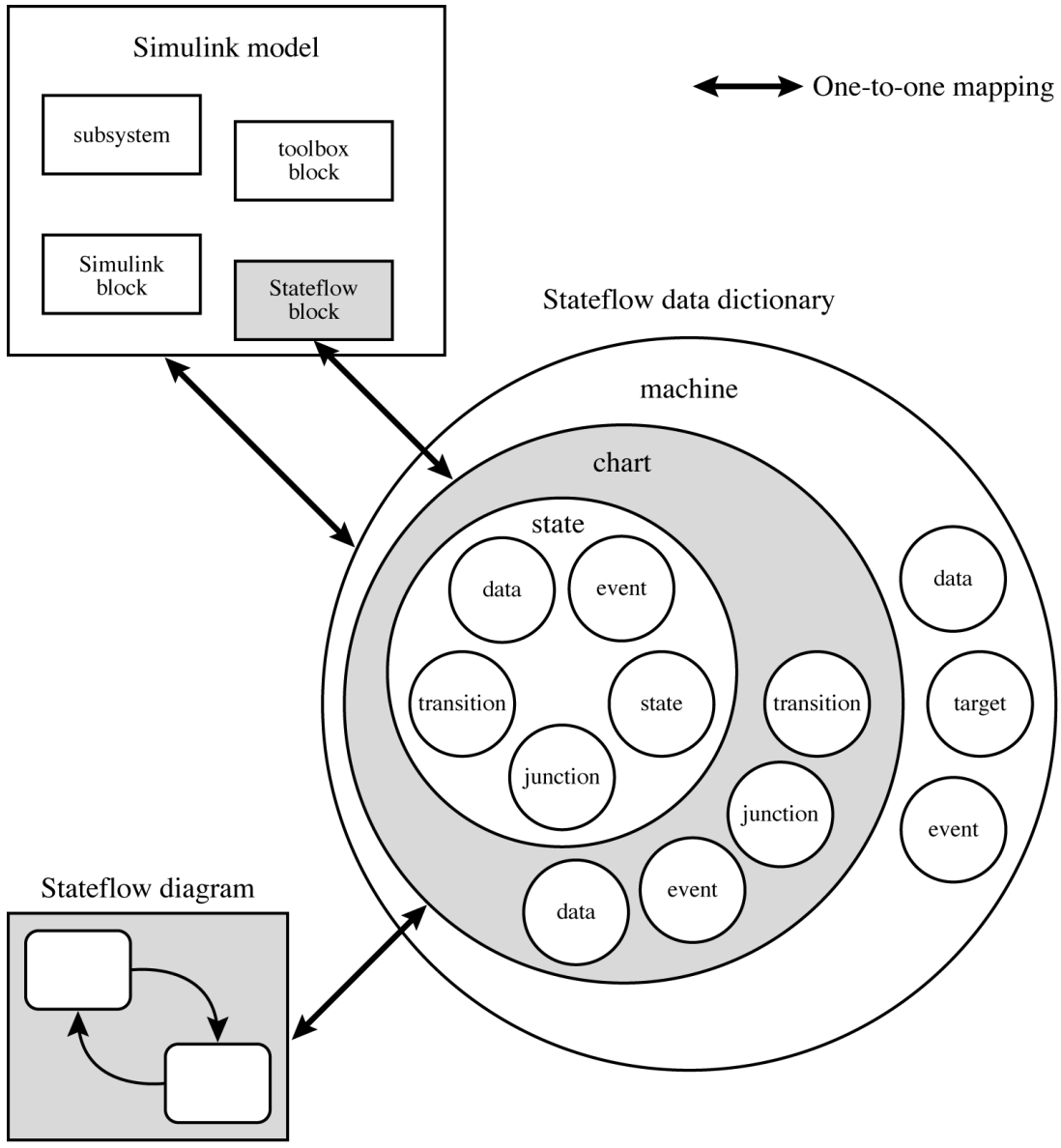
Anatomy of a Model and Machine

The Simulink Model and Stateflow Machine

The Stateflow machine is the collection of Stateflow blocks in a Simulink model. The Simulink model and Stateflow machine work seamlessly together. Running a simulation automatically executes both the Simulink and Stateflow portions of the model.

A Simulink model can consist of combinations of Simulink blocks, toolbox blocks, and Stateflow blocks (Stateflow diagrams). In Stateflow, the chart (Stateflow diagram) consists of a set of graphical (states, transitions, connective junctions, and history junctions) and nongraphical (event, data, and target) objects.

There is a one-to-one correspondence between the Simulink model and the Stateflow machine. Each Stateflow block in the Simulink model is represented in Stateflow by a single chart (Stateflow diagram). Each Stateflow machine has its own object hierarchy. The Stateflow machine is the highest level in the Stateflow hierarchy. The object hierarchy beneath the Stateflow machine consists of combinations of the graphical and nongraphical objects. The data dictionary is the repository for all Stateflow objects.



Stateflow scoping rules dictate where the types of nongraphical objects can exist in the hierarchy. For example, data and events can be parented by the

machine, the chart (Stateflow diagram), or by a state. Targets can only be parented by the machine. Once a parent is chosen, that object is known in the hierarchy from the parent downwards (including the parent's offspring). For example, a data object parented by the machine is accessible by that machine, by any charts within that machine, and by any states within that machine. The hierarchy of the graphical objects is easily and automatically handled for you by the graphics editor. You manage the hierarchy of nongraphical objects through the Explorer or the graphics editor **Add** menu.

Defining Stateflow Interfaces

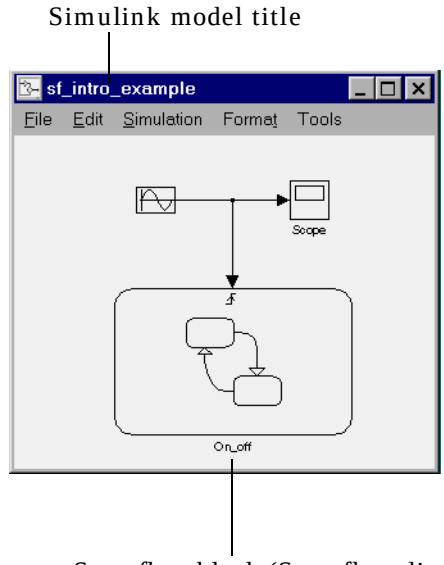
Each Stateflow block corresponds to a single Stateflow diagram. The Stateflow block interfaces to its Simulink model. The Stateflow block may interface to code sources external to the Simulink model (data, events, custom code).

Stateflow diagrams are event driven. Events can be local to the Stateflow block or can be propagated to and from Simulink and code sources external to Simulink. Data can be local to the Stateflow block or can be shared with and passed to the Simulink model and to code sources external to the Simulink model.

You must define the interface to each Stateflow block. Defining the interface for a Stateflow block can involve some or all of these tasks:

- Defining the Stateflow block update method
- Defining **Output to Simulink** events
- Adding and defining nonlocal events and nonlocal data within the Stateflow diagram
- Defining relationships with any external sources

In this example, the Simulink model titled `sf_intro_example` consists of a Simulink Sine Wave source block, a Simulink Scope sink block, and a single Stateflow block, titled `Example`.

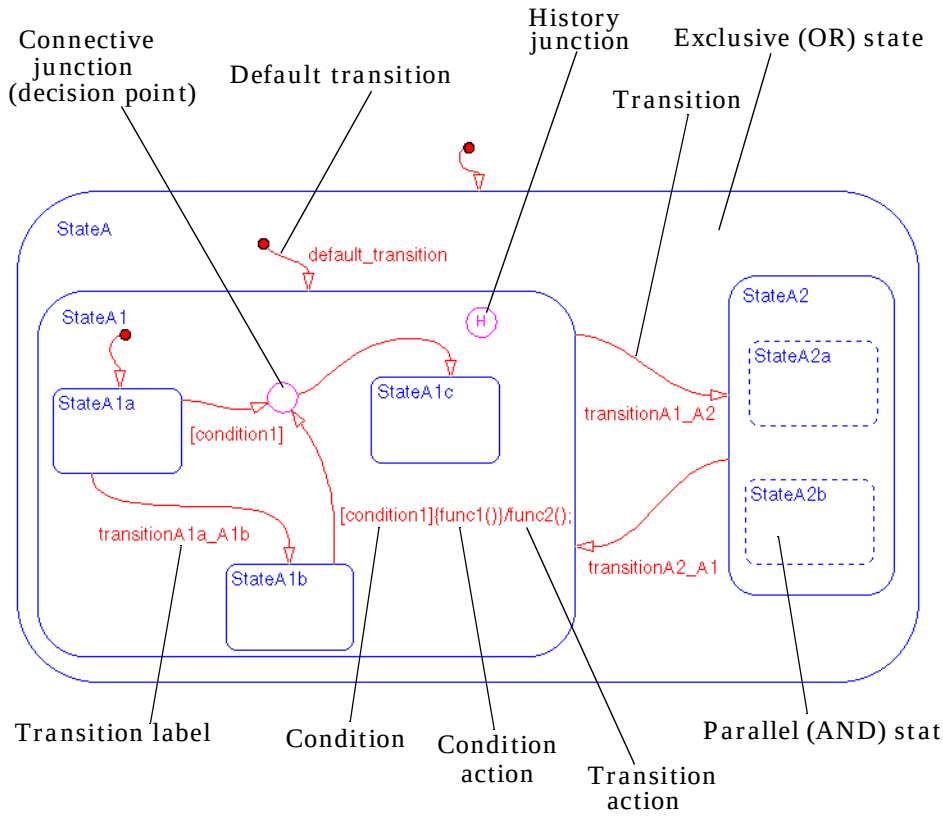


See “Defining the Stateflow Interface” on page 1-30 and Chapter 4, “Defining Stateflow Interfaces” for more information.

Stateflow Diagram Objects

This sample Stateflow diagram highlights some key graphical components. The sections that follow describe these graphical components as well as some nongraphical objects and related concepts in greater detail.

Figure 1-1: Graphical Components



States

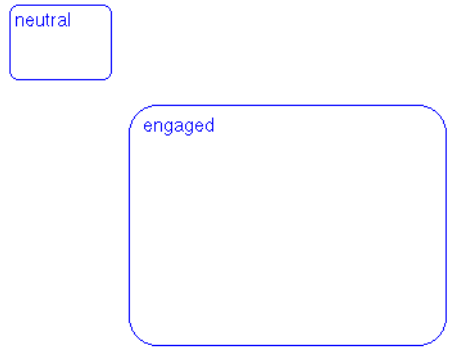
A *state* describes a mode of an event-driven system. The activity or inactivity of the states dynamically changes based on events and conditions.

Every state has a parent. In a Stateflow diagram consisting of a single state, that state's parent is the Stateflow diagram itself (also called the Stateflow diagram root). You can place states within other higher-level states. In the Graphical Components figure, `StateA1` is a child in the hierarchy to `StateA`.

A state also has history. History provides an efficient means of basing future activity on past activity.

States have labels that can specify actions executed in a sequence based upon action type. The action types are: `entry`, `during`, `exit`, and `on`.

In an automatic transmission example, the transmission can either be in neutral or engaged in a gear. Two states of the transmission system are `neutral` and `engaged`.

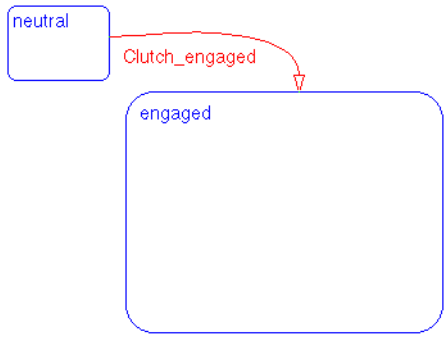


Stateflow provides two types of states: parallel (AND) and exclusive (OR) states. You represent parallelism with AND (parallel) states. The transmission example shows exclusive (OR) states. Exclusive (OR) states are used to describe modes that are mutually exclusive. The system is either in the `neutral` state or the `engaged` state at any one time.

Transitions

A *transition* is a graphical object that, in most cases, links one object to another. One end of a transition is attached to a source object and the other end to a destination object. The *source* is where the transition begins and the *destination* is where the transition ends. A *transition label* describes the circumstances under which the system moves from one state to another. It is always the occurrence of some event that causes a transition to take place. In the Graphical Components figure, the transition from `StateA1` to `StateA2` is labeled with the event `transitionA1_A2` that triggers the transition to occur.

Consider again the automatic transmission system. `clutch_engaged` is the event required to trigger the transition from `neutral` to `engaged`.



Events

Events are nongraphical objects and are thus not represented directly in the Graphical Components figure. Events drive the Stateflow diagram execution. All events that affect the Stateflow diagram must be defined. The occurrence of an event causes the status of the states in the Stateflow diagram to be evaluated. The broadcast of an event can trigger a transition to occur or can trigger an action to be executed. Events are broadcast in a top-down manner starting from the event's parent in the hierarchy.

Events are created and modified using the Stateflow Explorer. Events can be created at any level in the hierarchy. Events have properties such as a scope. The scope defines whether the event is:

- Local to the Stateflow diagram
- An input to the Stateflow diagram from its Simulink model
- An output from the Stateflow diagram to its Simulink model
- Exported to a (code) destination external to the Stateflow diagram and Simulink model
- Imported from a code source external to the Stateflow diagram and Simulink model

Data

Data are nongraphical objects and are thus not represented directly in the Graphical Components figure. Data objects are used to store numerical values for reference in the Stateflow diagram.

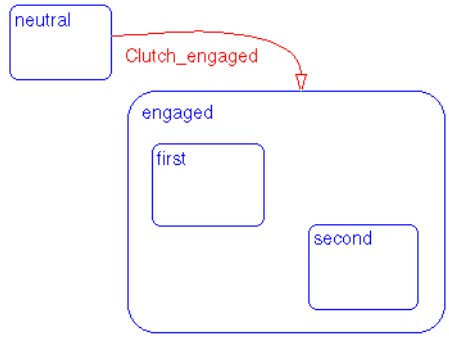
Data objects are created and modified using the Stateflow Explorer. Data objects can be created at any level in the hierarchy. Data objects have properties such as a scope. The scope defines whether the data object is:

- Local to the Stateflow diagram
- An input to the Stateflow diagram from its Simulink model
- An output from the Stateflow diagram to its Simulink model
- Non-persistent temporary data
- Defined in the MATLAB workspace
- A constant
- Exported to a (code) destination external to the Stateflow diagram and Simulink model
- Imported from a code source external to the Stateflow diagram and Simulink model

Hierarchy

Hierarchy enables you to organize complex systems by defining a parent and offspring object structure. A hierarchical design usually reduces the number of transitions and produces neat, manageable diagrams. Stateflow supports a hierarchical organization of states such that states can exist within other states. Stateflow represents state hierarchy with superstates and

substates.For example, this Stateflow diagram has a superstate that contains two substates.



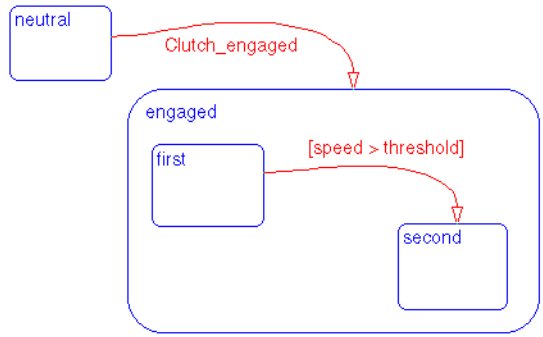
The engaged superstate contains the `first` and second substates. The engaged superstate is the parent in the hierarchy to the states `first` and second. When the event `clutch_engaged` occurs, the system transitions out of the `neutral` state to the engaged superstate. Transitions within the engaged superstate are intentionally omitted from this example for simplicity.

A transition out of a higher level, or *superstate*, also implies transitions out of any active substates of the superstate. Transitions can cross superstate boundaries to specify a substate destination. If a substate is active its parent superstate is also active.

Conditions

A *condition* is a Boolean expression specifying that a transition occurs, given that the specified expression is true. In the component summary Stateflow diagram, `[condition1]` represents a Boolean expression that must be true for the transition to occur.

In the automatic transmission system, the transition from `first` to second occurs if the Boolean condition, `[speed > threshold]` is true.

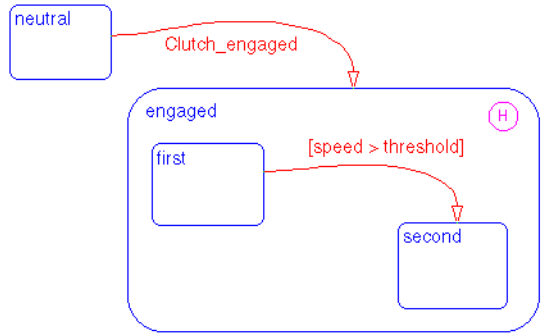


History Junction

History provides the means to specify the destination substate of a transition based on historical information. If a superstate with exclusive (OR) decomposition has a history junction, the transition to the destination substate is defined to be the substate that was most recently visited. A history junction applies to the level of the hierarchy in which it appears. The history junction overrides any default transitions. In the component summary Stateflow diagram, the history junction in `StateA1` indicates that when a transition to `StateA1` occurs, the substate that becomes active (`StateA1a`, `StateA1b`, or `StateA1c`) is based on which of those substates was most recently active.

In the automatic transmission system, history indicates that when `clutch_engaged` causes a transition from `neutral` to the `engaged` superstate,

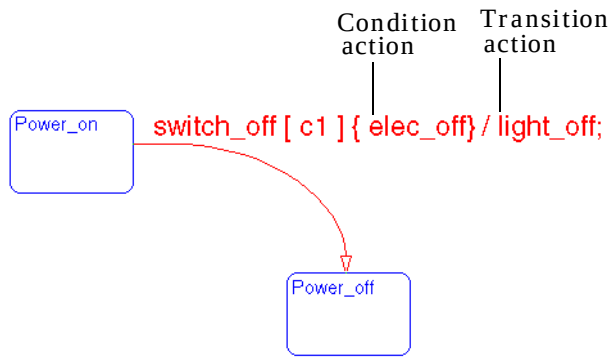
the substate that becomes active, either `first` or `second`, is based on which of those substates was most recently active.



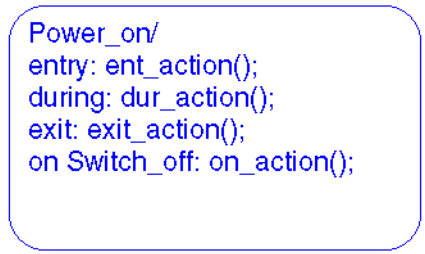
Actions

Actions take place as part of Stateflow diagram execution. The action can be executed either as part of a transition from one state to another or based on the activity status of a state. In the Graphical Components figure, the transition segment from `StateA1b` to the connective junction is labeled with a condition action (`func1()`) and a transition action (`func2()`). The semantics of how and why actions take place are discussed throughout the examples in Chapter 7, "Semantics."

Transitions can have *condition* actions and *transition* actions, as shown in this example:



States can have `entry`, `during`, `exit`, and `on` *event_name* actions. For example:



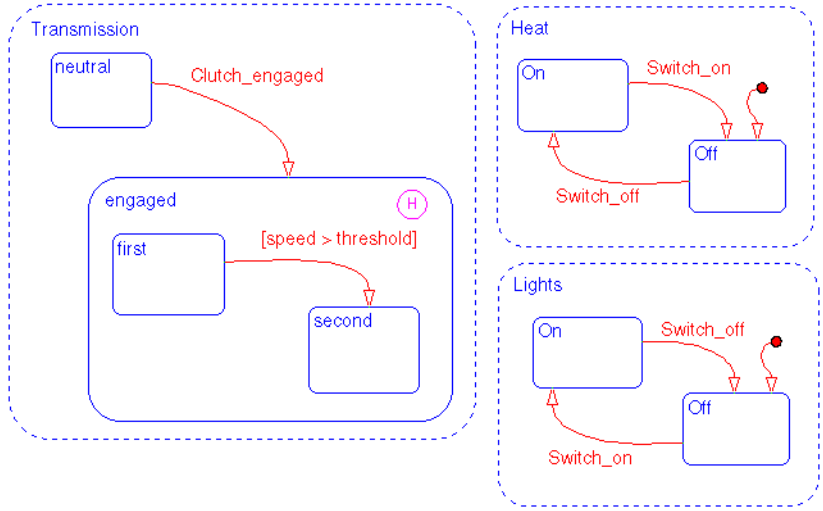
The *action language* defines the types of actions you can specify and their associated notations. An action can be a function call, an event to be broadcast, a variable to be assigned a value, etc.

Stateflow supports both Mealy and Moore finite state machine modeling paradigms. In the Mealy model, actions are associated with transitions, whereas in the Moore model they are associated with states. Stateflow supports state actions, transition actions, and condition actions. For more information, see the section titled "What Is an Action Language?" on page 6-42.

Parallelism

A system with *parallelism* has two or more states that can be active at the same time. The activity of each parallel state is essentially independent of other states. In the Graphical Components figure, `StateA2a` and `StateA2b` are parallel (AND) states. `StateA2` has parallel (AND) state decomposition.

For example, this Stateflow diagram has parallel superstate decomposition.



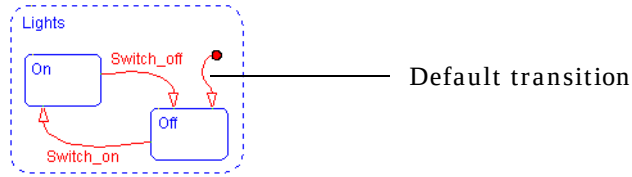
The transmission, heating, and light systems are parallel subsystems in a car. They exist in parallel and are physically independent of each other. There are many other parallel components in a car, such as the braking and windshield wiper subsystems.

You represent parallelism in Stateflow by specifying parallel (AND) state decomposition. Parallel (AND) states are displayed as dashed rectangles.

Default Transitions

Default transitions specify which exclusive (OR) state is to be active when there is ambiguity between two or more exclusive (OR) states at the same level in the hierarchy. In the Graphical Components figure, when `StateA` is active, by default `StateA1` is also active. Without the default transition to `StateA1`, there is ambiguity in whether `StateA1` or `StateA2` should be active.

In the `Lights` subsystem, the default transition to the `Lights.Off` substate indicates when the `Lights` superstate becomes active, the `Off` substate becomes active by default.

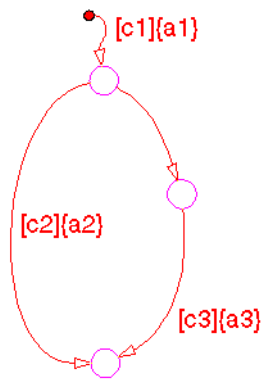


Default transitions specify which exclusive (OR) substate in a superstate the system enters by default, in the absence of any information. History junctions override default transition paths in superstates with exclusive (OR) decomposition.

Connective Junctions

Connective junctions are decision points in the system. A connective junction is a graphical object that simplifies Stateflow diagram representations and facilitates generation of efficient code. They provide alternative ways to represent desired system behavior. In the Graphical Components figure, the connective junction is used as a decision point for two transition segments that complete at `StateA1c`.

This example shows how connective junctions (displayed as small circles) are used to represent the flow of an `if` code structure.




```
if [c1]{
     a1
     if [c2]{
        a2
     }else if [c3]{
       a3
     }
}
```

Using Stateflow

Stateflow Typical Tasks

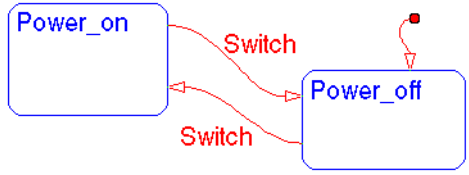
Using Stateflow typically involves:

- Creating or starting with a Simulink model
- Creating a Stateflow diagram
- Defining a Stateflow block interface
- Running a simulation
- Debugging
- Generating code

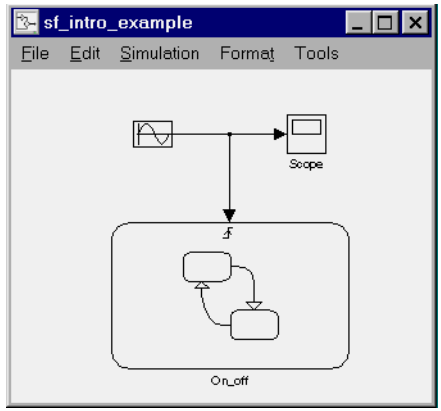
Depending upon your design approach, the details of the tasks and/or the sequence you do them in will vary.

Sample Solution

Here is a sample of the completed Stateflow diagram:



Here is a sample of the completed Simulink model:



When you simulate this model, the generation of the input event from Simulink, `Switch`, will toggle the activity of the states between `Power_on` and `Power_off`.

Creating a Simulink Model

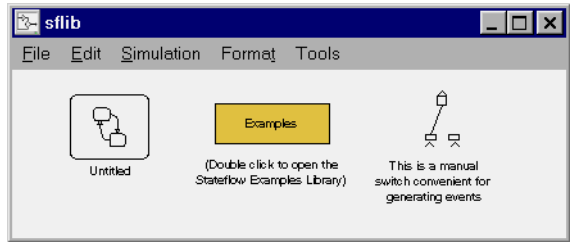
Opening the Stateflow model window is the first step towards creating a Simulink model with a Stateflow block. By default, an untitled Simulink model with an untitled, empty Stateflow block is created for you when you open the Stateflow model window. You can either start with the default empty model or copy the untitled Stateflow block into any Simulink model to include a Stateflow diagram in an existing Simulink model.

These steps describe how to create a Simulink model with a Stateflow block, label the Stateflow block, and save the model.

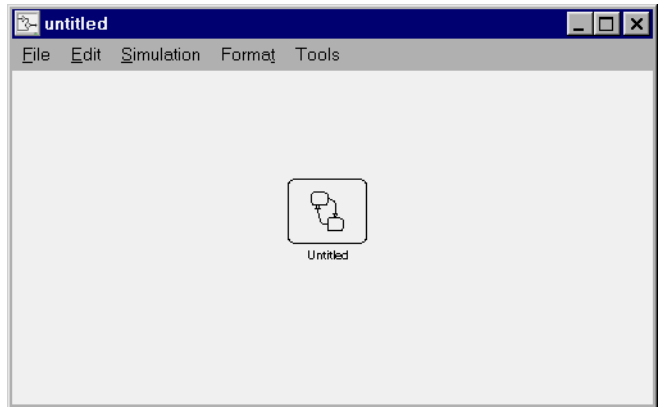
1 Display the Stateflow model window.

At the MATLAB prompt enter `stateflow`.

Here is the Stateflow model window:

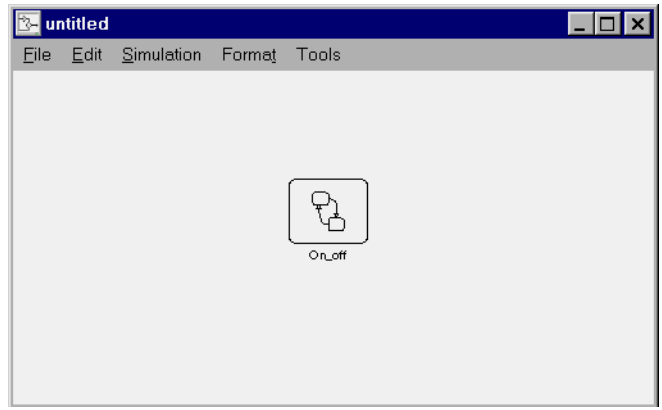

sflib
File Edit Simulation Format Tools
Untitled
Examples
(Double click to open the Stateflow Examples Library)
This is a manual switch convenient for generating events

The window consists of untitled Stateflow block icon, an Examples block, and a manual switch. This is the untitled Simulink model window with an untitled Stateflow block:



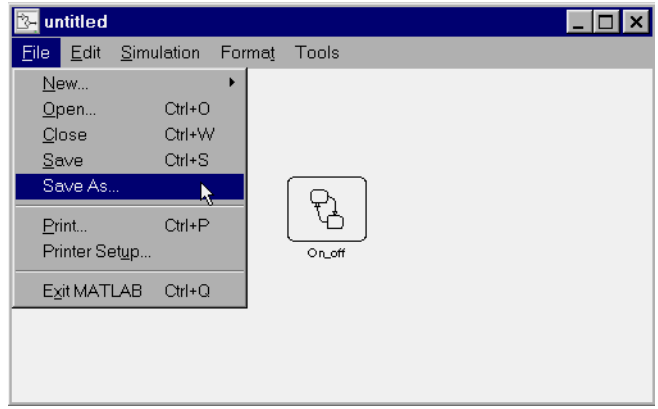
2 Label the Stateflow block.

Label the Stateflow block in the new untitled model by clicking in the text area and replacing the text “Untitled” with the text `On_off`.



3 Save the model.

Choose **Save As** from the **File** menu of the Simulink model window. Enter a model title.



You can also save the model by choosing **Save** or **Save As** from the Stateflow graphics editor **File** menu. Saving the model either from Simulink or from the graphics editor saves all contents of the Simulink model.

Creating a Stateflow Diagram

These steps describe how to create a simple Stateflow diagram using the graphics editor.

1 Invoke the graphics editor.

Double-click on the Stateflow block in the Simulink model window to invoke the graphics editor window.

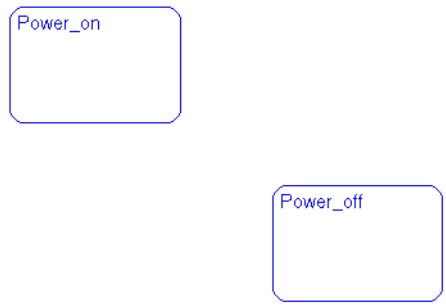
2 Create states.

Click on the **State** button in the toolbar. Click in the drawing area to place the state in the drawing area. Position the cursor over that state, click

the right mouse button, and drag to make a copy of the state. Release the right mouse button to drop the state at that location.

3 Label states.

Click on the ? character within each state to enter each state label. Label the states with these titles: `Power_on` and `Power_off`. Deselect the state to exit the edit. Your Stateflow diagram should look similar to this sample.

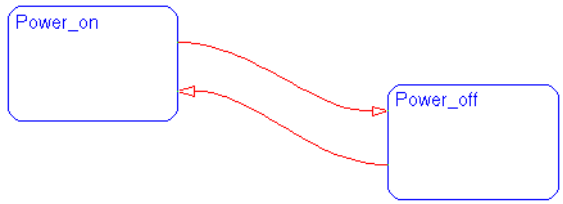


4 Create transitions.

Draw a transition starting from `Power_on` and ending at `Power_off`. Place the cursor at a straight portion of the border of the `Power_on` state. Click the border when the cursor changes to a crosshair. Without releasing the mouse button, drag the mouse to a straight portion on the border of the `Power_off` state. When the transition snaps to the border of the `Power_off` state,

release the mouse button. (The crosshair will not appear if you place the cursor on a corner, since corners are used for resizing.)

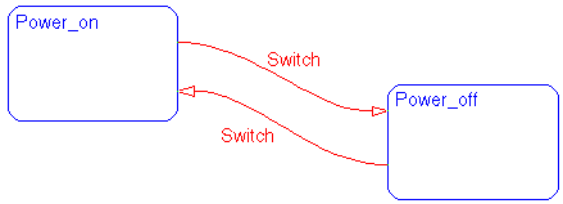
Draw another transition starting from `Power_off` and ending on `Power_on`. Your Stateflow diagram should look similar to this sample.



5 Label the transitions.

Click on the transition from `Power_on` to `Power_off` to select it. Click on the ? alongside the transition and enter the label `Switch`. Press the **Escape** key to deselect the transition label and exit the edit.

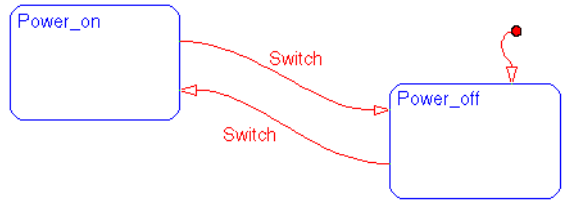
Label the transition from `Power_off` to `Power_on` with the same text, `Switch`. Your Stateflow diagram should look similar to this sample.



6 Add a default transition.

Click and release the mouse on the **Default Transition** button in the toolbar. Drag the mouse to a straight portion on the border of the `Power_off` state. Click and release the mouse when the arrowhead snaps to the border

of the `Power_off` state. Your Stateflow diagram should look similar to this sample.



7 Parse the Stateflow diagram.

Parsing the Stateflow diagram ensures that the notations you have specified are valid and correct.

Choose **Parse Diagram** from the **Tools** menu of the graphics editor to parse the Stateflow diagram. Informational messages are displayed in the MATLAB command window. Any error messages are displayed in red. If no red error messages appear the parse operation is successful and the text 'Done' is displayed.

See "Parsing" on page 9-11 for more information on parsing.

For More Information

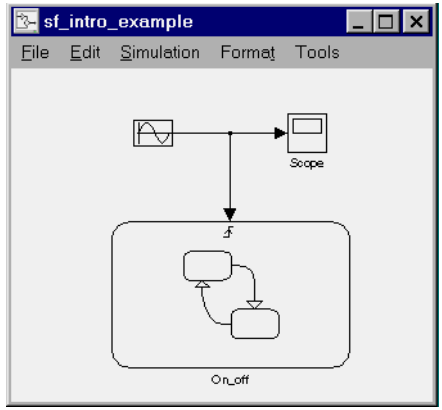
For more information on creating Stateflow diagrams using the graphics editor see "Using the Graphics Editor."

Defining the Stateflow Interface

In this example, there are no external code sources to include in the interface definition. These steps show how to define the Stateflow block interface in the Simulink model.

1. Add and define input events within the Stateflow diagram.
   a. Choose **Explore** from the graphics editor **Tools** menu to invoke the Explorer.
   b. Double-click on the machine name (same as the Simulink model name) in the **Object Hierarchy** list.
   c. Click on the `On_off` chart entry in the Object Hierarchy list.
   d. Select **Event** from the **Add** menu.
   e. Enter `Switch` in the **Name** field of the **Event properties** dialog box.
   f. Select **Input from Simulink** as the **Scope** value.
   g. Select **Rising Edge** as the **Trigger** type.
   h. Click on the **Close** button to apply the changes and close the window.
   i. Choose **Close** from the Explorer **File** menu to close the Explorer.
2. Make connections in the Simulink model between other blocks and the Stateflow block.
   a. Enter `Simulink` in the MATLAB command window to invoke Simulink.
   b. Add a Sine Wave block (located in the Simulink Sources block library) and connect it to the input trigger port of the Stateflow block.

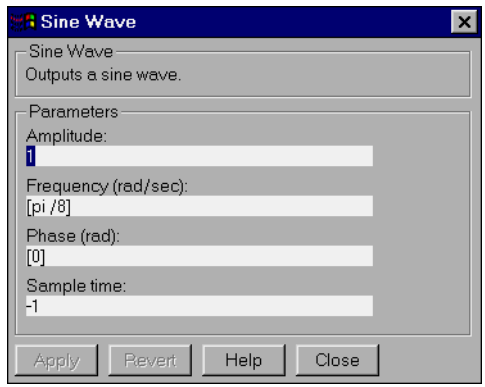
c Add a Scope block (located in the Simulink Sinks block library) and connect it to the Sine Wave block output as well. Your model should look similar to this.



Click on the **Close** button to apply the changes and close the dialog box.

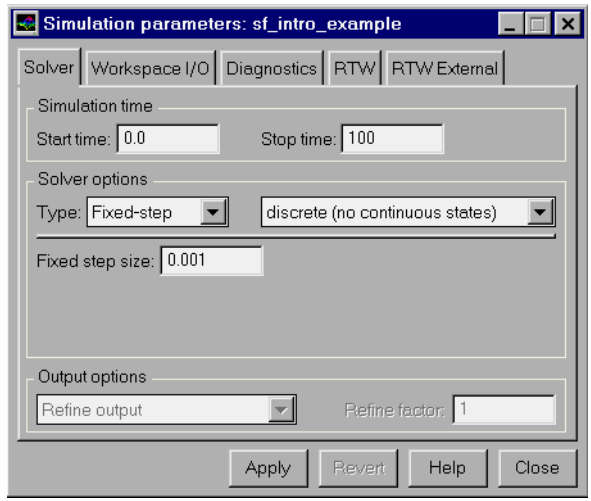
3 Define Simulink parameters.

a Double-click on the Sine Wave block and edit the parameters as shown in this example dialog box.



Click on the **Close** button to apply the changes and close the dialog box.

b Choose **Parameters** from the **Simulation** menu of the Simulink model window and edit the values to match the values in this dialog box.



For More Information

For more information on Stateflow interfaces beyond what is discussed here, see “Defining Stateflow Interfaces”.

Running a Simulation

These steps show how to run a simulation.

1 Ensure the Stateflow diagram and any display blocks such as the Scope block, are open.

Double-click on the `On_off` Stateflow block to display the Stateflow diagram. Double-click on the Scope block to display the output of the Sine wave.

2 Select **Open Simulation Target** from the graphics editor **Tools** menu. Ensure that the checkbox to **Enable Debugging/Animation** is checked. Click on the **Close** button to apply the change and close the window.

3 Select **Debug** from the graphics editor **Tools** menu. Ensure that the **Enabled** radio button under **Animation** is checked to enable Stateflow

diagram animation. Click on the **Close** button to apply the change and close the window.

4 Choose **Start** from the graphics editor **Simulation** menu to start a simulation of the model.

   By default the S-function is the simulation target for any Stateflow blocks. Informational and error messages related to the S-function code generation for Stateflow blocks are displayed in the MATLAB command window. Once the simulation is started and the target is built, the graphics editor becomes read-only (frozen).

   The input from the Sine block is defined as the **Input from Simulink** event `Switch`. When the simulation starts the Stateflow diagram is animated reflecting the state changes triggered by the input sine wave. Each input event of `Switch` toggles the model between the `Power_off` and `Power_on` state.

5 Choose **Stop** from the graphics editor **Simulation** menu to stop a simulation. Once the simulation is stopped the graphics editor becomes writeable (unfrozen).

Debugging

The Stateflow Debugger supports functions like single stepping, animating, and running up to a designated breakpoint and then stopping.

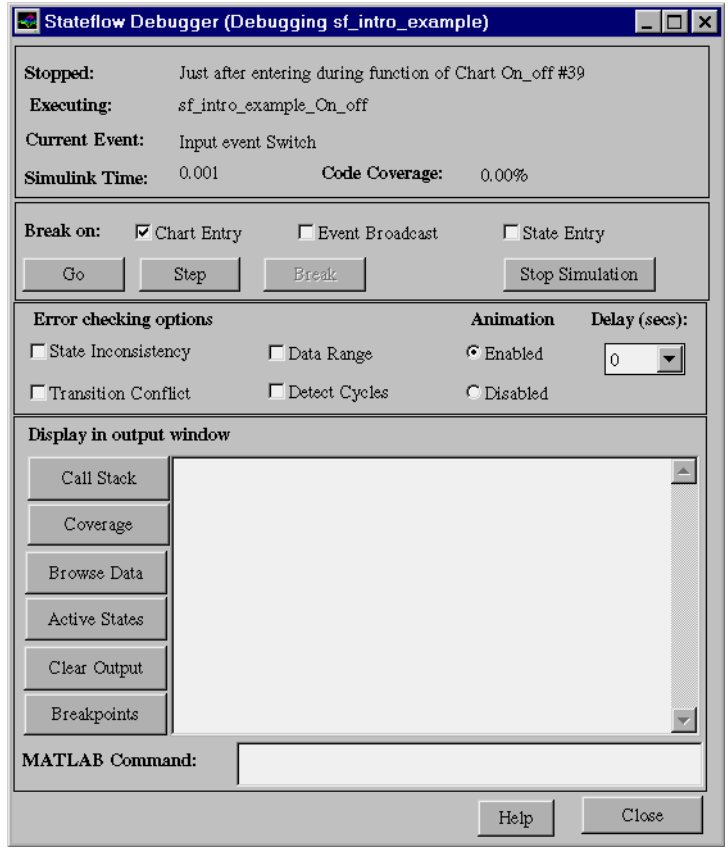
These steps show how to step through the simulation using the Debugger.

1 Display the Debugger by choosing **Debug** from the **Tools** menu of the graphics editor.

2 Click on the **Break on**: **Chart Entry** checkbox to specify you want the Debugger to stop the simulation execution when the chart is entered.

3 Click on the **Go** button to start the simulation. Informational and error messages related to the S-function code generation for Stateflow blocks are displayed in the MATLAB command window. When the target is built, the

graphics editor becomes read-only (frozen) and the Debugger window will be updated and look similar to this:



4 Click on the **Step** button to proceed one step at a time through the simulation. Take note of the top portion of the Debugger window where this information is displayed:

- Where the simulation is stopped
- What is executing
- The current event
- The simulation time
- The current code coverage percentage

Watch the graphics editor window as you click on the **Step** button to see each transition and state animated when it is executing. After both `Power_off` and `Power_on` have become active by stepping through the simulation, the code coverage indicates 100%.

5 Choose **Stop** from the graphics editor **Simulation** menu to stop a simulation. Once the simulation is stopped the graphics editor becomes writeable (unfrozen).

6 Click on the **Close** button in the Debugger window.

7 Choose **Close** from the **File** menu in the Simulink model window.

For More Information

See “Debugging” for more information beyond the debugging topics in this section.

Generating Code

If you have the Stateflow Coder product, you can generate stand-alone code suitable to target for a particular processor. Stateflow generates simulation code (`sfun` target) for the `On_off` Stateflow block.

The `On_off` Stateflow block and its Simulink model is not designed or intended to generate stand-alone code for a specific target processor.

For More Information

See “Generating Code” for more information on code generation.

2

Using the Graphics Editor

The Graphics Editor Main Window 2-2

Shortcut Menus 2-10

Using the Graphics Editor 2-15

Working with Charts (Stateflow Diagrams) 2-18

Working with States 2-24

Working with Transitions 2-31

Working with Junctions 2-38

Main Editor Menus 2-42

Defaults File 2-50

The Graphics Editor Main Window

Overview

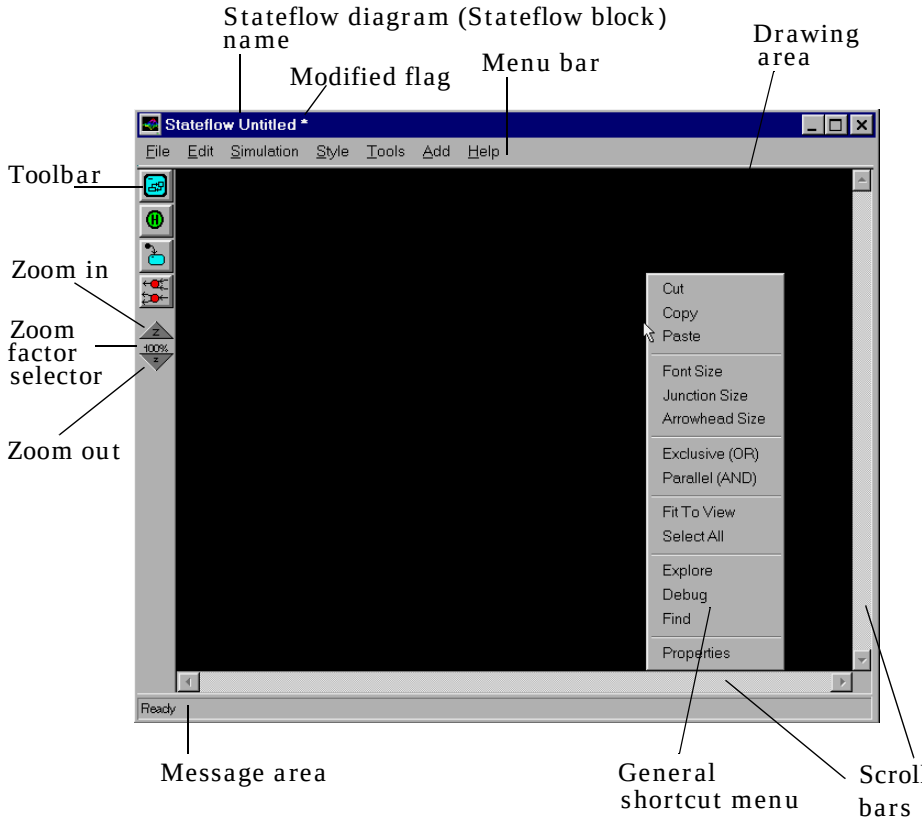
Using the graphics editor you can visually model complex reactive systems based on *finite state machine* theory and statechart formalisms. The graphics editor is designed with productivity in mind. The user interface features a number of productivity boosts (see "Productivity Boosts" on page 2-7).

See "Using Stateflow" on page 1-22 for a quick tutorial on using the graphics editor.

Main Window Components

The main window consists of a toolbar, pull-down menus, and a drawing area. The window is titled with the name of the current Stateflow diagram. The toolbar on the left-hand side contains object buttons that correspond to

drawing objects. Using these object buttons, you create your Stateflow diagrams.



Drawing Area

The drawing area is the region of the main window where graphical objects can be placed. The size of the drawing area is increased by resizing the main window.

Message Area

Informational messages about what object or operation is active in the editor (graphics editor focus messages) are displayed in the message area in the lower left-hand side of the main window.

Zooming Options

These are the zoom control options:

- Click on the **Zoom In** button to make a quick zoom in to the closest discrete zoom factor.
- Click on the **Zoom Out** button to make a quick zoom out to the closest discrete zoom factor.
- Use the **Zoom Factor Selector** to choose a discrete zoom factor percentage from a menu. Click on the percentage value to display the discrete percentage value pulldown menu. Click on the down arrow to display the list of percentages. Click on the percentage to change the zoom factor to that percentage. The default percentage is initially 100% and is subsequently calculated as the closest discrete value to the current zoom factor.
- Double-click on the **Zoom Factor Selector** or choose **Fit to View** from any shortcut menu to apply the maximum zoom that includes all selected objects. If there is no selection list, all objects are fit to the view.
- Press the **F** key to apply the maximum zoom that includes all selected objects. Press the spacebar to fit all objects to the view. If there is no selection list, all objects are fit to the view.
- Use the Zoom virtual scrollbar to choose a zoom percentage from a sliding scale. Click on the percentage value and drag the mouse vertically to use the virtual scrollbar. Dragging the mouse upward increases the zoom factor. Dragging the mouse downward decreases the zoom factor. Release the mouse button to make a zoom factor selection. Alternatively, right-clicking and dragging on the percentage value resizes while you are dragging.

Scroll Bars

Clicking on the arrows in the scroll bar adjusts your view of the drawing area to view objects that may be outside of the current drawing area.

Stateflow Diagram (Stateflow Block) Name

The name of a new, untitled Stateflow diagram is `Untitled`. The name appears in the upper left-hand side of the main window. Use the **Save Model As...** menu item on the **File** menu to specify a Stateflow diagram name. You can also change the Stateflow block name in the Simulink model. Once you have saved the Stateflow diagram, the new name appears.

Modified Flag

An asterisk following the Stateflow diagram/Stateflow block name indicates that an unsaved change has been made to the Stateflow diagram.

Menu Bar

The menu choices provide the ability to operate on the graphical Stateflow diagram objects and perform operations on the Stateflow diagram itself. The menu bar consists of these menus:

- **File**
- **Edit**
- **Simulation**
- **Style**
- **Tools**
- **Add**
- **Help**

Toolbar

You can select from these object buttons on the toolbar:

Name	Button Icon	Description
State		Use a State to depict a mode of the system.
History junction		Use a history junction to indicate, when entering this level in the hierarchy, that the last state that was active becomes the next state to be active.
Default transition		Use a default transition to indicate, when entering this level in the hierarchy, which state becomes active by default (in the absence of a history junction).
Connective junction		One use of a connective junction is to handle situations where transitions out of one state into two or more states are taken based on the same event but guarded by different conditions.

Productivity Boosts

These features of the graphics editor increase productivity by providing shortcut methods for performing common graphic editor tasks:

- Virtual scrollbars; see “Virtual Scrollbar” on page 2-7.
- Double-click fit to view; see “Zooming Options” on page 2-4.
- F key and spacebar zooming operations; see “Zooming Options” on page 2-4
- Shortcut menus; see “Shortcut Menus” on page 2-10.
- Escape key; see “Canceling Operations” on page 2-15.
- Multiple object mode; see “Object Button Modes” on page 2-15.

Virtual Scrollbar

A virtual scrollbar enables you to set a value. When you move the mouse over a menu item with a virtual scrollbar, the cursor changes to a line with a double arrowhead. Virtual scrollbars are either vertical or horizontal. The direction is indicated by the positioning of the arrowheads. Click on the menu item and drag the mouse either horizontally or vertically to change the value.

These menu items on the various shortcut menus (see “Shortcut Menus” on page 2-10) have virtual scrollbars:

- Font Size
- Junction Size
- Arrowhead Size

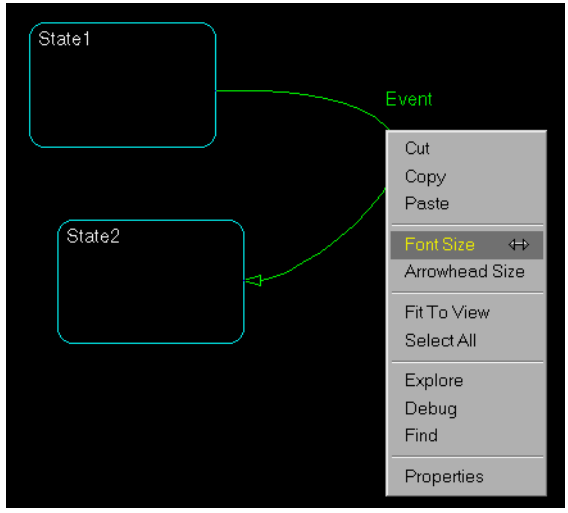
The **Zoom Factor Selector** also has a virtual scrollbar.

Example: Adjusting Font Size

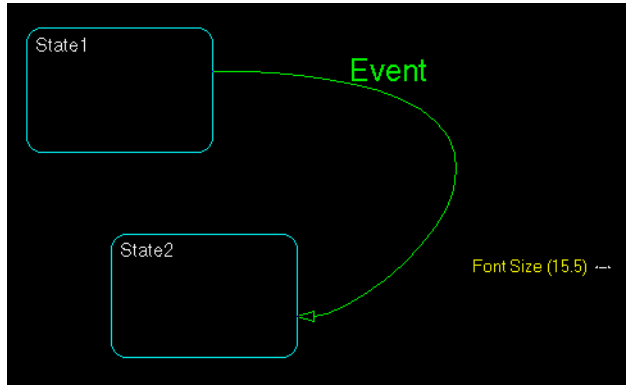
Font size is adjusted through the shortcut menus using a virtual scrollbar. To adjust font size with the virtual scrollbar:

1 Right-click on the object(s) whose font you want to change to display the shortcut menu.

2 Place the cursor over the **Font Size** menu item.



3 Click and drag horizontally on the **Font Size** menu item. The shortcut menu disappears and a numeric size is displayed.



4 Drag the mouse to the left to decrease the font size or to the right to increase the font size.

If you are using the general shortcut menu, the font size of all labels on any selected objects is adjusted. If you are using one of the object-specific shortcut menus, the font size of all labels on any selected objects of that type is adjusted.

Shortcut Menus

Overview

Shortcut menus provide a quick way to perform common functions. The shortcut menus are context sensitive. Clicking the right mouse button displays the shortcut menu for the object the cursor is positioned over. There are four shortcut menus that are displayed depending upon where the cursor is positioned:

- **General** (not over any object) (see "General Shortcut Menu" on page 2-12)
- **State** (over a state) (see "State Shortcut Menu" on page 2-13)
- **Transition** (over a transition) (see "Transition Shortcut Menu" on page 2-14)
- **Junction** (over a junction) (see "Junction Shortcut Menu" on page 2-14)

Menu Options Available on All Shortcut Menus

All of the shortcut menus have these menu options:

Menu Choice	Description
Cut	Moves selected object(s) from the drawing area to the Stateflow clipboard.
Copy	Copies selected object(s) to the Stateflow clipboard.
Paste	Pastes object(s) from the Stateflow clipboard to the drawing area at the current mouse location.

Menu Choice	Description
Fit to View	Applies panning and the maximum zoom that includes all selected objects. If there is no selection list, all objects are fit to the view. (see "Zooming Options" on page 2-4)
Select All	Selects all object(s) in the Stateflow diagram.
Explore	Invokes the Stateflow Explorer.
Debug	Invokes the Debugger.
Find	Invokes the Stateflow Finder.
Properties	Displays the selected object's properties dialog box.

See these sections for descriptions of the specific behavior: "General Shortcut Menu" on page 2-12, "State Shortcut Menu" on page 2-13, "Transition Shortcut Menu" on page 2-14, and "Junction Shortcut Menu" on page 2-14.

Menu options specific to an object type are available only from that object's shortcut menu. The **Font Size** menu option is the only exception; font size is modified via a virtual scrollbar on the **General**, **State**, and **Transition** shortcut menus and is modified to a discrete value via the graphics editor **Style** menu.

General Shortcut Menu

Move the cursor to an open area of the drawing area and click the right mouse button to display the **General** shortcut menu.

The **General** shortcut menu provides these additional general options:

Menu Choice	Description
Font Size	Changes the font size of all labels on any selected objects using a virtual scrollbar.
Junction Size	Changes the junction size on any selected junctions using a virtual scrollbar.
Arrowhead Size	Changes the arrowhead size on any selected destination object(s) using a virtual scrollbar.
Exclusive (OR)	Sets the parent decomposition on any selected states to be exclusive (OR).
Parallel (AND)	Sets the parent decomposition on any selected states to be parallel (AND).

State Shortcut Menu

Move the cursor over a state or its border and click the right mouse button to display the **State** shortcut menu. The **State** shortcut menu provides these state-specific options:

Menu Choice	Description
Font Size	Changes the font size of all labels on any selected states using a virtual scrollbar.
Arrowhead Size	Changes the arrowhead size on any selected states using a virtual scrollbar.
Exclusive (OR)	Sets the parent decomposition on any selected states to be exclusive (OR).
Parallel (AND)	Sets the parent decomposition on any selected states to be parallel (AND).
Show/Hide	Toggles between showing or hiding the selected superstate contents.
Group/Ungroup	Toggles between grouping or ungrouping the selected superstate contents.

Transition Shortcut Menu

Move the cursor over a transition and click the right mouse button to display the **Transition** shortcut menu. The **Transition** shortcut menu provides these transition-specific options:

Menu Choice	Description
Font Size	Changes the font size of all labels on any selected transitions using a virtual scrollbar.
Arrowhead Size	Changes the arrowhead size on any selected transitions using a virtual scrollbar. Changing one of the incoming arrowheads of an object causes all incoming arrowheads to that object to be adjusted to the same size.

Junction Shortcut Menu

Click the right mouse button within a junction or on its border to display the **Junction** shortcut menu. The **Junction** shortcut menu provides these junction-specific options:

Menu Choice	Description
Junction Size	Changes the junction size on any selected junctions using a virtual scrollbar.
Arrowhead Size	Changes the arrowhead size on any selected junctions using a virtual scrollbar.
History	Changes any selected junction to a history junction.
Connective	Changes any selected junction to a connective junction.

Using the Graphics Editor

Object Button Modes

All object buttons (state, history junction, default transition, and connective junction) have two modes of operation to place objects in the drawing area. In either mode, the cursor changes to a + character when the editor is in an object button mode.

- Single object mode

  Click on the object button and drag to place one object of that type in the drawing area. Alternatively, click on the object button, release, and click in the drawing area to choose the location for that object. Pressing the **Escape** key cancels any operation.

- Multiple object mode

  Double-click on the object button. Additional clicks in the drawing area create and place multiple objects of that type. Click on the object button to release the button. Pressing the **Escape** key cancels any operation.

Canceling Operations

Pressing the **Escape** key cancels the current operation and deselects any objects on the selection list.

Selecting and Deselecting Objects

Once an object is in the drawing area, you need to select it to make any changes or additions to that object. To select an object, click on it. When an object is selected, it is highlighted in the color set as the selection color.

To select multiple objects, click the left mouse button and drag the selection rubberband so that the grayed frame encompasses or touches the objects you want to select. Once all objects are within the rubberband, release the left mouse button. All objects or portions of objects within the rubberband are selected.

Simultaneously pressing the **Shift** key and clicking on an object either adds that object to the selection list if it was deselected or deselects the object if it is on the selection list. This is useful to select objects within a state without selecting the state itself.

To select all objects in the Stateflow diagram, choose **Select All** from the **Edit** menu or the right mouse button shortcut menu.

Simultaneously, pressing the **Shift** key and doing a rubberband selection selects objects touched by the rubberband if they are deselected and deselects objects touched by the rubberband if they are selected.

Pressing the **Escape** key deselects all selected objects. Pressing the **Escape** key again brings the Simulink model (the parent of the Stateflow diagram) forward.

Cutting and Pasting Objects

You can cut one or more objects from the drawing area or cut and then paste the object(s) as many times as you like. You can cut and paste objects from one Stateflow diagram to another. The Stateflow clipboard contains the most recently cut selection list of objects. The object(s) are pasted in the drawing area location closest to the current mouse location.

To cut an object, select the object and choose **Cut** from either:

- The **Edit** menu on the main window
- The right mouse button shortcut menu

Pressing the **Ctrl** and **X** keys simultaneously is the keyboard equivalent to the **Cut** menu item. The Stateflow clipboard contains the most recent deleted selection list of objects.

To paste the most recently cut selection list of objects, choose **Paste** from either:

- The **Edit** menu on the main window
- The right mouse button shortcut menu

Pressing the **Ctrl** and **V** keys simultaneously is the keyboard equivalent to the **Paste** menu item.

Copying Objects

To copy and paste an object in the drawing area, select the object(s), click and hold the right mouse button down, and drag to the desired location in the drawing area. This operation also updates the Stateflow clipboard.

Alternatively, to copy from one Stateflow diagram to another, choose the **Copy** and then **Paste** menu items from either:

- The **Edit** menu on the Stateflow graphics editor window
- Any right mouse button shortcut menu

Pressing the **Ctrl** and **C** keys simultaneously is the keyboard equivalent to the **Copy** menu item. States that contain other states (superstates) can be grouped together. (See “Grouping and Ungrouping States” on page 2-26.) Once selected, you can copy a group as though it was a single object. All objects in the group are copied.

Working with Charts (Stateflow Diagrams)

Overview

In Simulink, each chart (Stateflow diagram) is represented by a Stateflow block. A Simulink model can consist of combinations of Simulink blocks, toolbox blocks, and Stateflow blocks (Stateflow diagrams). The collection of Stateflow blocks in the Simulink model is the Stateflow machine.

In Stateflow, the chart (Stateflow diagram) consists of a set of graphical (states, transitions, connective junctions, and history junctions) and nongraphical (events and data) objects. Within a chart, this level is called the root of the hierarchy.

What Can You Do with Charts?

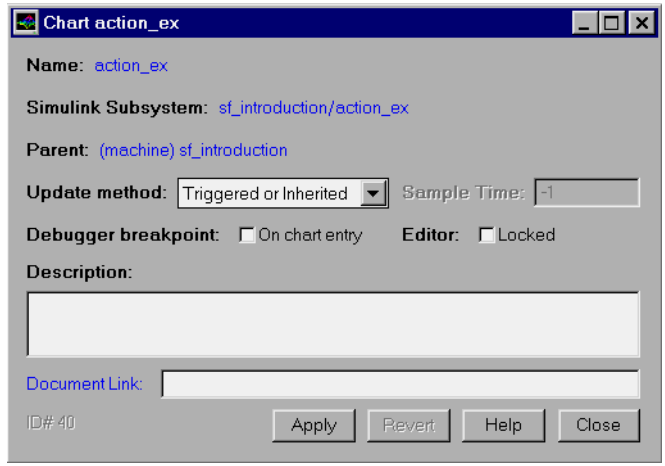
One of the most significant chart-specific tasks is defining the interface between the chart and the Simulink model and/or external sources. See Chapter 4, “Defining Stateflow Interfaces” for detailed information.

At the chart level you can:

- Invoke the chart properties dialog box to:
  - Use the hypertext link to the Stateflow diagram, the Simulink subsystem, or the parent’s property dialog box
  - Specify which method will be used to awaken the chart
  - Specify debugger breakpoints
  - Add a description
  - Link to a document
- Explicitly print a chart

Editing Chart (Stateflow Diagram) Properties

Click the right mouse button in an open area of the Stateflow diagram to display the **General** shortcut menu. This is the **Chart properties** dialog box.



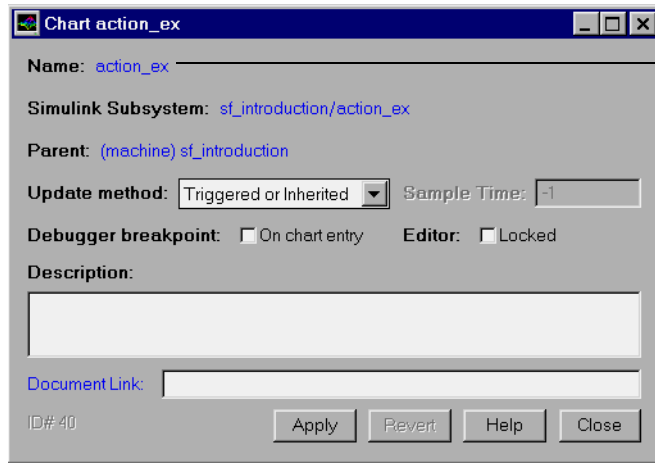
This table lists and describes the chart object fields.

Field	Description
Name	Stateflow diagram name; read-only; click on this hypertext link to bring the chart to the foreground.
Simulink Subsystem	Simulink subsystem name; read-only; click on this hypertext link to bring the Simulink subsystem to the foreground.
Parent	Parent name; read-only; click on this hypertext link to display the parent's property dialog box.
Update Method	Choose from **Triggered or Inherited**, **Sampled**, or **Continuous.**
Sample Time	If **Update Method** is **Sampled**, enter a sample time.
Debugger breakpoint	Click on the checkbox to set a debugging breakpoint **On chart entry.**
Editor	Click on the **Locked** checkbox to mark the Stateflow diagram as read-only and prohibit any write operations.
Description	Textual description/comment.
Document Link	Enter a web URL address or a general MATLAB command. Examples are: `www.mathworks.com`, `mailto:email_address`, `edit/spec/data/speed.txt`.

Click on the **Apply** button to save the changes. Click on the **Revert** button to cancel any changes and return to the previous settings. Click on the **Close** button to save the changes and close the dialog box. Click on the **Help** button to display the Stateflow online help in an HTML browser window.

Chart Hyperlinks Example

This is an example of a chart properties dialog box:



Hypertext link to the chart.

Click on the hypertext in the **Name** field. The chart is brought to the foreground.

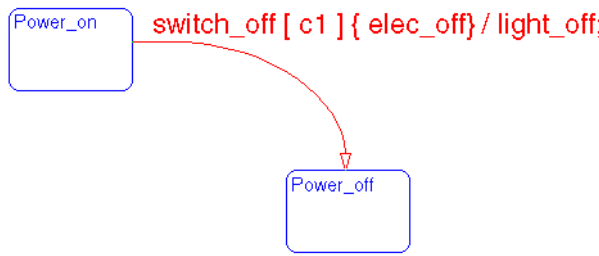


Chart Update Methods

The chart is awakened when an event occurs. You can choose from these methods of having the chart awakened, entered, and executed:

- Triggered/Inherited

  This is the default update method.

  - Triggered

    The Stateflow block is explicitly triggered by a signal originating from a connected Simulink block. The edge trigger can be set to **Rising**, **Falling**, **Either**, or **Function Call**.

  - Inherited

    The Stateflow block inherits (implicitly) triggers from the Simulink model. These implicit events are the sample times (discrete-time or continuous) of the Simulink signals providing inputs to the chart. The sample times are determined by Simulink to be consistent with the rates of all the incoming signals.

- Sampled

  Simulink will awaken (sample) the Stateflow block at the rate you specify. An implicit event is generated by Simulink at regular time intervals corresponding to the specified rate. The sample time is in the same units as the Simulink simulation time. Note that other blocks in the Simulink model may have different sample times.

- Continuous

  Simulink will awaken (sample) the Stateflow block at each step in the simulation, as well as at intermediate time points that may be requested by the Simulink solver.

See “Defining the Interface to External Sources” on page 4-25 and *Using Simulink* for more information.

Printing Large Stateflow Diagrams

To print complete Stateflow diagrams that are larger than the Stateflow graphics editor display area:

1. Press the **Escape** key to deselect all Stateflow diagram objects.
2. Double-click on the **Zoom Factor Selector** to fit the Stateflow diagram to the page.
3. Choose **Print...** from the **File** menu or enter a print command from the MATLAB command line. Pressing the **Ctrl** and **P** keys simultaneously is the keyboard equivalent to the **Print** menu item.

   See the section titled “File Menu” on page 2-43 for more information on printing from the editor window. See `sfprint` in the “Reference” chapter for more information about printing from the command line.

Working with States

What Can You Do with States?

You can:

- Specify decomposition
- Create/delete
- Label
- Group/ungroup
- Show/hide
- Resize
- Change font size
- Change arrowhead size
- Move
- Invoke the state properties dialog box to:
  - Use the hypertext link to the state or the parent
  - Specify debugger breakpoints
  - Add or modify the label
  - Add a description
  - Link to a document

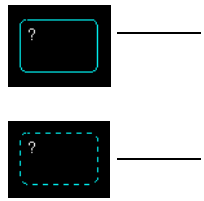
Specifying Decomposition

States that contain other states are called *superstates*. A superstate has either Parallel (AND) or Exclusive (OR) decomposition. The decomposition of the Stateflow diagram can be set if it contains at least one state. Specify state decomposition by selecting the superstate and choosing either the **Parallel (AND)** or **Exclusive (OR)** menu items from the right mouse button shortcut menu. In the case of the Stateflow diagram, display the shortcut menu by clicking the right mouse button in an area outside the borders of all states. The

Stateflow diagram decomposition reverts to exclusive (OR) decomposition if all states are deleted.

Creating/Deleting States

To create one state at a time, click on the **State** button in the toolbar, and then click on a location in the drawing area.



To create multiple states, double-click on the **State** button in the toolbar. The button is now in multiple object mode. Click anywhere in the drawing area to place a state in the drawing area. Additional clicks in the drawing area create additional states. Click on the **State** button or press the **Escape** key to cancel the operation.

To delete a state, select it and choose **Cut** (**Ctrl**+**X**) from the **Edit** or any shortcut menu or press the **Delete** key.

Labeling States

The ? character is the default empty label for states. Click on the ? to edit the label. Press the **Return** key to continue the label on the next line. Deselect the object to exit the edit.

State labels have this general format:

```
name/
entry:
during:
exit:
on event_name:
```

The keywords `entry` (shorthand `en`), `during` (shorthand `du`), `exit` (shorthand `ex`), and `on` identify actions associated with the state. Each keyword is optional and positionally independent. You can specify none, all, or some of them. The colon after each keyword is required. The slash following the state name is optional as long as it is followed by a carriage return.

If you enter the name and slash followed directly by an action or actions (without the `entry` keyword), the action(s) are interpreted as `entry` action(s). This shorthand is useful if you are only specifying `entry` actions.

See the section titled “Labeling a State” on page 6-13 for more information.

Grouping and Ungrouping States

States that contain other states are called *superstates*. All of the states contained in a superstate can be grouped together for ease of movement, copying, and uniform scaling when resizing. Once selected, you can move and resize a group as though it was a single object. When you move a grouped superstate, the contents are automatically hidden for efficiency in the moving operation.

Double-click on or within the superstate border to group all of the states contained in a superstate. A grouped superstate is represented graphically by a thicker border. The appearance varies slightly depending on the type of decomposition (exclusive (OR) or parallel (AND)) and the platform. The contents of a grouped superstate are always highlighted in a light shade of gray.

To ungroup a grouped superstate, double-click on or within the grouped superstate border. You need to ungroup the superstate to select objects within the superstate.

Showing and Hiding States

Hiding the contents of a superstate simplifies the Stateflow diagram if that level of detail is not necessary at a particular moment.

To hide the contents of a superstate, move the cursor over the superstate, press the right-most mouse button to display the **State** shortcut menu and choose **Show/Hide**. Hiding the contents of an ungrouped state causes the state to be grouped. Choosing the **Show/Hide** menu item toggles between the two choices.

Resizing States

To resize a state, drag one of the state's corners. When the cursor is over a corner, it appears as a double-ended arrow (PC only; cursor appearance will vary on other platforms). If you resize a grouped superstate, the contents of the grouped superstate are resized appropriately. The font size of state labels is proportionally resized.

Changing Font Size

You can adjust font size through the shortcut menus using a virtual scrollbar. To adjust font size with the virtual scrollbar:

1. Select the state(s) whose label font size you want to change.
2. Right-click on a selected state whose font you want to change to display the shortcut menu.
3. Place the cursor over the **Font Size** menu item.
4. Click and drag horizontally on the **Font Size** menu item. The shortcut menu disappears and a numeric size is displayed.
5. Drag the mouse to the left to decrease the font size or to the right to increase the font size.

   The font size of all labels on any selected states is adjusted.

Changing Arrowhead Size

The arrowhead size is a property of the destination object. Changing one of the incoming arrowheads of a state causes all incoming arrowheads to that state to be adjusted to the same size. The arrowhead size of any selected states is adjusted.

To adjust arrowhead size from the **State** shortcut menu:

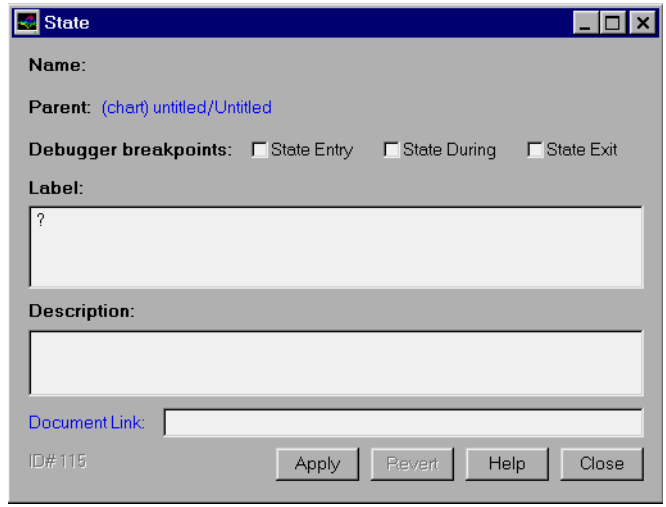
1. Select the state(s) whose incoming arrowhead size you want to change.
2. Place the cursor over a selected state, click the right mouse button to display the shortcut menu. Place the cursor over **Arrowhead Size.** The shortcut menu disappears and a numeric size is displayed.
3. Click and drag horizontally to the left to decrease the arrowhead size or to the right to increase the arrowhead size. The numeric display adjusts to show the currently selected arrowhead size.

Moving States

To move a state, select, drag, and release it in a new position. If you move a grouped superstate, the superstate and its contents are moved.

Editing State Properties

Select the state, click the right mouse button to display the **State** shortcut menus and, choose **Properties**. This is the **State properties** dialog box.



This table lists and describes the state object fields.

Field	Description
Name	Stateflow diagram name; read-only; click on this hypertext link to bring the state to the foreground.
Parent	Parent of this state; the / character indicates the Stateflow diagram is the parent; read-only; click on this hypertext link to bring the parent to the foreground.
Debugger breakpoints	Click on the checkboxes to set debugging breakpoints on state `entry`, `during`, or `exit` actions.
Label	The state's label. See the section titled "Labeling a State" on page 6-13 for more information on valid label formats.
Description	Textual description/comment.
Document Link	Enter a web URL address or a general MATLAB command. Examples are: `www.mathworks.com`, `mailto:email_address`, `edit/spec/data/speed.txt`.

Click on the **Apply** button to save the changes. Click on the **Revert** button to cancel any changes and return to the previous settings. Click on the **Close** button to save the changes and close the dialog box. Click on the **Help** button to display the Stateflow online help in an HTML browser window.

Working with Transitions

What Can You Do with Transitions?

You can:

- Create/delete
- Edit attach points
- Label
- Change font size
- Change arrowhead size
- Move labels
- Invoke the transition properties dialog box to:
  - Use the hypertext link to transition's source, destination, or parent
  - Specify debugger breakpoints
  - Add or modify the label
  - Add a description
  - Link to a document

Creating/Deleting Transitions

Place the cursor at a straight portion of the border of the source state. Click the border when the cursor changes to a cross-hair. Drag and release on a straight portion of the border of the destination state when the transition changes from a dotted line to a solid line. The solid line indicates the transition is in position to be attached. The cross-hair will not appear if you place the cursor on a corner, since corners are used for resizing.

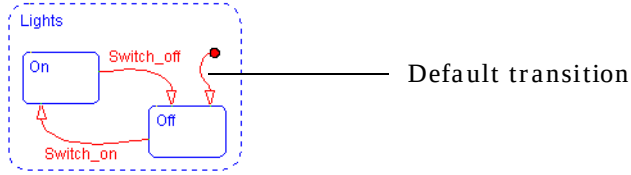
Use the same procedure to create transitions to and from junctions. The only difference is that transitions can be connected to any point on the junction. (Junctions are resized through the **Junction** shortcut menu.)

To delete a transition, select it and choose **Cut** (**Ctrl + X**) from the **Edit** menu or any shortcut menu or press the **Delete** key.

What Is a Default Transition?

The default transition object is a transition with a destination, but no source object. Default transitions specify which exclusive (OR) state is to be active when there is ambiguity between two or more exclusive (OR) states at the same level in the hierarchy. Default transitions also specify that a junction should be entered by default.

In the `Lights` subsystem, the default transition to the `Lights.Off` substate indicates when the `Lights` superstate becomes active, the `Off` substate becomes active by default.



Default transitions specify which exclusive (OR) substate in a superstate the system enters by default, in the absence of any information. History junctions override default transition paths in superstates with exclusive (OR) decomposition.

Creating Default Transitions

Click on the **Default transition** button in the toolbar , and click on a location in the drawing area close to the state or junction you want to be the destination for the default transition. Drag the mouse to the destination object to attach the default transition. The size of end point of the default transition is proportional to the arrowhead size. Default transitions can be labeled just like other transitions. See the section titled “Labeling of Default Transitions” on page 6-24 for an example.

Editing Transition Attach Points

Place the cursor over an attach point until it changes to a small circle. Click and drag the mouse to move the attach point; release to drop the attach point. You can edit both the source and destination attach points of a transition.

The appearance of the transition changes from a solid to a dashed line when editing a destination attach point. Once you attach the transition to a destination, the dashed line changes to a solid line. The appearance of the transition changes to a default transition when editing a source attach point.

To delete a transition, select it and choose **Cut** (**Ctrl**+**X**) from the **Edit** or any shortcut menu or, press the **Delete** key.

Labeling Transitions

The ? character is the default empty label for transitions. Transitions and default transitions follow the same labeling format. Select the transition to display the ? character. Click on the ? to edit the label.

Transition labels have this general format:

```
event [condition]{condition_action}/transition_action
```

Specify, as appropriate, relevant names for `event`, `condition`, `condition_action`, and `transition_action`. Transitions do not have to have labels. You can specify some, all, or none of the parts of the label.

Label Field	Description
`event`	Event that causes the transition to be evaluated.
`condition`	Defines what, if anything, has to be true for the condition action and transition to take place.
`condition_action`	If the condition is true, the action specified executes and completes.
`transition_action`	After a valid destination is found and the transition is taken, this action executes and completes.

To apply and exit the edit, deselect the object.

See these sections in Chapter 6, “Notations” for more information:

- “Transitions” on page 6-17
- “Action Language” on page 6-42

Valid Labels

Labels can consist of any alphanumeric and special character combination, with the exception of embedded spaces. Labels cannot begin with a numeric character. The length of a label is not restricted. Carriage returns are allowed in most cases. Within a condition, you must specify an ellipsis (...) to continue on the next line.

Changing Font Size

Font size is adjusted through the shortcut menus using a virtual scrollbar. To adjust font size with the virtual scrollbar:

1. Select the transition(s) whose label font size you want to change.
2. Right-click on a selected transition whose font you want to change, to display the shortcut menu.
3. Place the cursor over the **Font Size** menu item.
4. Click and drag horizontally on the **Font Size** menu item. The shortcut menu disappears and a numeric size is displayed.
5. Drag the mouse to the left to decrease the font size or to the right to increase the font size.

   The font size of all labels on any selected transitions is adjusted.

Changing Arrowhead Size

The arrowhead size is a property of the destination object. Changing one of the incoming arrowheads of an object causes all incoming arrowheads to that object to be adjusted to the same size. The arrowhead size of any selected transitions, and any other transitions ending at the same object, is adjusted.

To adjust arrowhead size from the **Transition** shortcut menu:

1. Select the transition(s) whose arrowhead size you want to change.
2. Place the cursor over a selected transition, click the right mouse button to display the shortcut menu. Place the cursor over **Arrowhead Size.** The shortcut menu disappears and a numeric size is displayed.
3. Click and drag horizontally to the left to decrease the arrowhead size or to the right to increase the arrowhead size. The numeric display adjusts to show the currently selected arrowhead size.

Moving Transition Labels

You can move transition labels to make the Stateflow diagram more readable. To move a transition label, click on and drag the label to the new location and then release the mouse button.

If you mistakenly click and release the left mouse button on the label, you will be in edit mode. Press the **Escape** key to deselect the label and try again.

Editing Transition Properties

Select a transition and click the right mouse button on that transition's border to display the **Transition** shortcut menu. Choose **Properties** to display the **Transition properties** dialog box.

This table lists and describes the transition object fields.

Field	Description
Source	Source of the transition; read-only; click on the hypertext link to bring the transition source to the foreground.
Destination	Destination of the transition; read-only; click on the hypertext link to bring the transition destination to the foreground.
Parent	Parent of this state; read-only; click on the hypertext link to bring the parent to the foreground.
Debugger breakpoints	Click on the checkboxes to set debugging breakpoints either when the transition is tested for validity or when it is valid.
Label	The transition's label. See the section titled "Labeling a Transition" on page 6-18 for more information on valid label formats.
Description	Textual description/comment.
Document Link	Enter a web URL address or a general MATLAB command. Examples are: `www.mathworks.com`, `mailto:email_address`, `edit/spec/data/speed.txt`.

Click on the **Apply** button to save the changes. Click on the **Revert** button to cancel any changes and return to the previous settings. Click on the **Close** button to save the changes and close the dialog box. Click on the **Help** button to display the Stateflow Online Help in an HTML browser window.

Working with Junctions

What Can You Do with Junctions?

You can:

- Create/delete
- Change size
- Change arrowhead size
- Move
- Invoke the junction properties dialog box to:
  - Hyperlink to the junction's parent.
  - Add a description.
  - Link to a document.

Creating/Deleting Junctions

To create one junction at a time, click on a **Junction** button in the toolbar and click on the desired location for the junction in the drawing area. To create multiple junctions, double-click on the **Junction** button in the toolbar. The button is now in multiple object mode. Click anywhere in the drawing area to place a junction in the drawing area. Additional clicks in the drawing area create additional junctions. Click on the **Junction** button or press the **Escape** key to cancel the operation.

You can choose from these types of junctions:

Name	Button Icon	Description
Connective junction		One use of a connective junction is to handle situations where transitions out of one state into two or more states are taken based on the same event but guarded by different conditions.
History junction		Use a history junction to indicate, when entering this level in the hierarchy, that the last state that was active becomes the next state to be active.

Changing Size

To adjust the junction size from the **Junction** shortcut menu:

1. Select the junction(s) whose size you want to change. The size of any selected junctions is adjusted.
2. Place the cursor over a selected junction, click the right mouse button to display the shortcut menu and place the cursor over **Junction Size**.
3. Click and drag horizontally to the left to decrease the junction size or to the right to increase the junction size. The numeric display adjusts to show the currently selected junction size.

Changing Arrowhead Size

The arrowhead size is a property of the destination object. Changing one of the incoming arrowheads of a junction causes all incoming arrowheads to that junction to be adjusted to the same size. The arrowhead size of any selected junctions is adjusted.

To adjust arrowhead size from the **Junction** shortcut menu:

1 Select the junction(s) whose incoming arrowhead size you want to change.

2 Place the cursor over a selected junction, click the right mouse button to display the shortcut menu. Place the cursor over **Arrowhead Size.** The shortcut menu disappears and a numeric size is displayed.

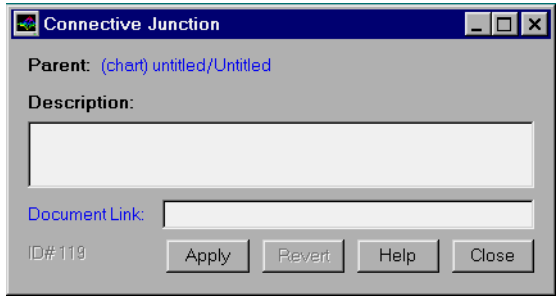
3 Click and drag horizontally to the left to decrease the arrowhead size or to the right to increase the arrowhead size. The numeric display adjusts to show the currently selected arrowhead size.

Moving a Junction

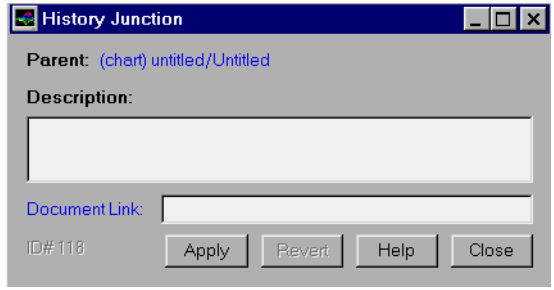
To move a junction, select, drag, and release it in a new position.

Editing Junction Properties

Select a junction, click the right mouse button on that junction to display the **Junction** shortcut menu. Choose **Properties** to display the **Connective Junction properties** dialog box.



This is the history junction properties dialog box.



This table describes the junction object fields.

Field	Description
Parent	Parent of this state; read-only; click on the hypertext link to bring the parent to the foreground.
Description	Textual description/comment.
Document Link	Enter a web URL address or a general MATLAB command. Examples are: `www.mathworks.com`, `mailto:email_address`, `edit/spec/data/speed.txt`.

Click on the **Apply** button to save the changes. Click on the **Revert** button to cancel any changes and return to the previous settings. Click on the **Close** button to save the changes and close the dialog box. Click on the **Help** button to display the Stateflow online help in an HTML browser window.

Main Editor Menus

Overview

The menu choices provide the ability to operate on the graphical Stateflow diagram objects and perform operations on the Stateflow diagram itself. The menu bar consists of these menus:

- **File**
- **Edit** (see "Edit Menu" on page 2-45)
- **Simulation** (see "Simulation Menu" on page 2-45)
- **Style** (see "Style Menu" on page 2-46)
- **Tools** (see "Tools Menu" on page 2-47)
- **Add** (see "Add Menu" on page 2-48)
- **Help** (see "Help Menu" on page 2-49)

File Menu

The **File** menu provides these options:

Menu Choice	Description
New Model (Ctrl + N)	Creates and displays a new untitled Simulink model and an untitled Stateflow block.
Open Model... (Ctrl + O)	Displays a navigational file manager dialog box you use to choose the Simulink model to open.
Save Model (Ctrl + S)	Saves the current Simulink model and any Stateflow blocks within the model.
Save Model As...	Displays a navigational dialog box for you to specify the directory and filename for the Simulink model.
Close (Ctrl + W)	Closes the current Stateflow diagram.
Close All Charts	Closes all open and/or iconified Stateflow diagrams in the Simulink model.
Print... (Ctrl P)	Prints the graphical portion of the current Stateflow diagram.
Print As	Displays a submenu of file format choices.
Printer Setup	Displays the platform-specific printer setup dialog box. (PC-only)
Exit MATLAB (Ctrl Q)	Closes all open Stateflow diagrams and models. Exits Stateflow, Simulink, and MATLAB environments.

Print As Submenu

The **Print As** submenu of the **File** menu provides these printing options:

Menu Choice	Description
PostScript	Saves the Stateflow diagram as a PostScript file.
Color PostScript	Saves the Stateflow diagram as a color PostScript file.
Encapsulated PostScript	Saves the Stateflow diagram as an encapsulated PostScript file.
Tiff	Saves the Stateflow diagram as a `tiff` file.
Jpeg	Saves the Stateflow diagram as a `jpeg` file.
Bitmap to Clipboard	Copies the Stateflow diagram to the clipboard (PC platform only).

Selection of one of these choices generates a file of the specified type in the directory defined as the MATLAB working directory. Enter `help cd` in the MATLAB command window to display help on how to change the working directory.

See the section titled “Printing Large Stateflow Diagrams” on page 2-23 for information on printing large Stateflow diagrams.

Edit Menu

The **Edit** menu provides these options:

Menu Choice	Description
Cut (Ctrl +X)	Moves selected object(s) from the drawing area to the Stateflow clipboard.
Copy (Ctrl + C)	Copies selected object(s) to the Stateflow clipboard.
Paste (Ctrl + V)	Pastes object(s) from the Stateflow clipboard to the drawing at the current mouse location.
Select All (Ctrl + A)	Selects all object(s) in the Stateflow diagram.

Simulation Menu

The **Simulation** menu provides these options:

Menu Choice	Description
Start (Ctrl + T)	Start the simulation.
Pause	Pause the simulation.
Parameters... (Ctrl + E)	Displays the **simulation parameters** dialog box.

See the *Using Simulink* document for more information on simulation parameters.

Style Menu

Using the **Style** menu you can change the color of these objects:

- States
- State labels
- Transitions
- Transition labels
- Junctions
- Chart

All objects of that type are redisplayed in the color you choose. You can also change the object selection color from this menu.

The color submenus of the **Style** menu provide these color options:

- Black
- Blue
- Green
- Cyan
- Red
- Magenta
- Yellow
- White

Using the **Style** menu you can also change the font size of any selected text.

Tools Menu

The **Tools** menu provides these options:

Menu Choice	Description
Explore (Ctrl+R)	Displays the Stateflow Explorer.
Debug (Ctrl+D)	Displays the Stateflow Debugger.
Find (Ctrl+F)	Displays the Stateflow Finder.
Parse	Parse all Stateflow diagrams in the current machine. Results of the parse operation are displayed in a pop-up window and in the MATLAB command window.
Parse Diagram	Parse the current Stateflow diagram. Results of the parse operation are displayed in a pop-up window and in the MATLAB command window.
Open Simulation Target	Displays the **Target Configuration sfun properties** dialog box.
Open RTW Target	Displays the **Target Configuration rtw properties** dialog box.

Add Menu

The **Add** menu provides these options:

Menu Choice	Description
Event	Choose a scope from the submenu; creates a new untitled event at the machine level in the object hierarchy (use the Explorer to modify the parent); displays an untitled **Event properties** dialog box.
Data	Choose a scope from the submenu; creates a new untitled data object at the machine level in the object hierarchy (use the Explorer to modify the parent); displays an untitled **Data properties** dialog box.
Target	Creates a new untitled target at the machine level in the object hierarchy (a target's parent must be a machine); displays an untitled **Target Configuration properties** dialog box.

In all cases, the machine is the object's parent by default. Use the Explorer to modify the parent of event or data objects. Use the Explorer to delete any of these objects.

See these sections for more information: "Example: Getting Started Adding Events and Data" on page 5-16 and "Stateflow Explorer User Interface" on page 5-5.

For more information on the **Event**, **Data**, and **Target Configuration property** dialog boxes, see these sections:

- "Event Properties Dialog Box" on page 5-9
- "Data Properties Dialog Box" on page 5-12
- "Adding and Specifying a Stand-Alone Target" on page 9-16

Help Menu

The **Help** menu provides these options:

Menu Choice	Description
Topics (HTML)	Displays Stateflow online help in an HTML browser. An online version of the *Stateflow User's Guide* is one of the options available from the online help.
Help Desk (HTML)	Displays the MATLAB Help Desk in an HTML browser.
About Stateflow	Displays a pop-up window with software version information.

Defaults File

What Is the Defaults File?

The defaults file contains default settings for:

- Machines
- Charts
- Targets
- States
- Transitions
- Junctions
- Data
- Events

What Do You Need to Know about the Defaults File?

The file is called `defaults` (no file extension) and is located in the directory listed by `sf('Root')` that is created as part of the installation process. You can edit this file to modify and customize the default values. You can add defaults for any user-writeable property.

The file is executed at initialization time when Stateflow is invoked. The file can be re-executed on demand from the MATLAB command line interface by entering `sfsave('defaults')`.

Machine Defaults

The machine defaults are:

Parameter	Default
`chart.name`	NULL
`chart.creator`	Unknown
`chart.version`	none

Defining Color

Colors are defined using the three-element matrix [red green blue] RGB notation. The matrix notation maps to the following colors. This table lists some representative RGB color definitions.

Color	Red	Green	Blue
Black	0	0	0
White	1	1	1
Red	1	0	0
Green	0	1	0
Blue	0	0	1
Yellow	1	1	0
Magenta	1	0	1
Cyan	0	1	1

Chart Defaults

The chart defaults are:

Parameter	Default
`chart.name`	Untitled
`chart.stateFontName`	Helvetica
`chart.transitionFontName`	Helvetica
`chart.stateColor`	[0 1 1] (Cyan)
`chart.stateLabelColor`	[1 1 1] (White)
`chart.transitionColor`	[1 1 0] (Yellow)

Parameter	Default
chart.transitionLabelColor	[1 1 0] (Yellow)
chart.groupedColor	[0.752941 0.752941 0.752941]
chart.junctionColor	[1 0 1] (Magenta)
chart.selectionColor	[0 1 0] (Green)
chart.animationColor	[1 0 0] (Red)
chart.chartColor	[0 0 0] (Black)

Target Defaults

The target defaults are:

Parameter	Default
target.name	Untitled
target.description	NULL
target.customCode	NULL
target.codeCommand	sfc
target.codeFlags	-sfun-debug-consistency-range-conflict
target.makeCommand	mex -v
target.simulationTarget	0

State Defaults

The state defaults are:

Parameter	Default
state.fontSize	9.0
state.arrowSize	8.0
state.visible	1

Transition Defaults

The transition defaults are:

Parameter	Default
transition.fontSize	9.0
transition.arrowSize	8.0
transition.visible	1

Junction Defaults

These junction defaults are defined in the defaults file.

Parameter	Default
junction.arrowSize	8.0
junction.visible	1

Data Defaults

These data object defaults are defined in the defaults file.

Parameter	Default
data.name	data
data.range	[-Inf +Inf]

Event Defaults

These event object defaults are defined in the defaults file.

Parameter	Default
event.name	NULL

3

Events and Data

Overview 3-2

Events 3-5

Implicit Local Events 3-20

Data 3-22

Overview

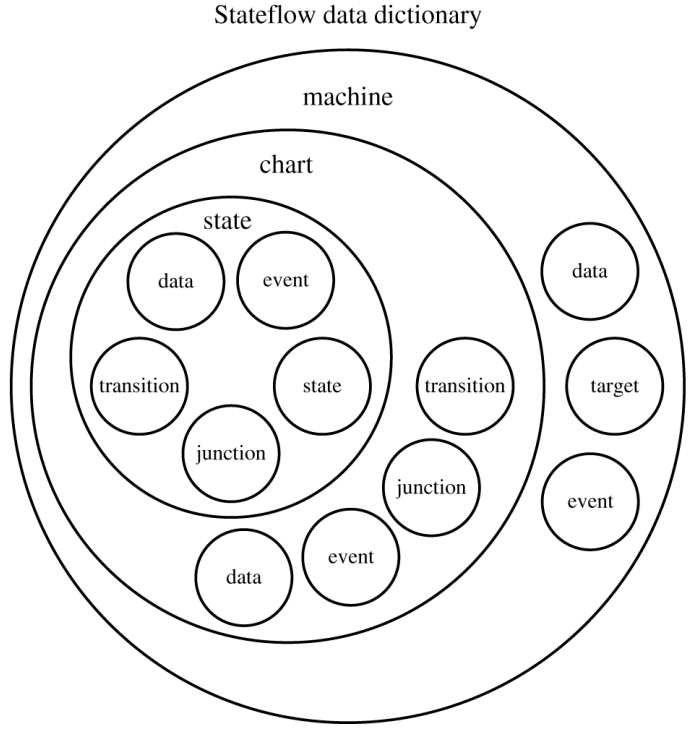
Events and Data

Events drive the Stateflow diagram execution. The occurrence of an event causes the status of the states in the Stateflow diagram to be evaluated. Data objects are used to store numerical values for reference in the Stateflow diagram. Events and data are nongraphical objects that are created using either the Stateflow Explorer (see “Stateflow Explorer User Interface” on page 5-5) or the graphics editor Add menu (see “Add Menu” on page 2-48).

You can define parents for events and data. Being able to define parents (hierarchy) and scope provides simulation and prototyping flexibility and efficient generated code. Most important of the benefits is the ability to optimize event processing through directed event broadcasting. Directed event broadcasting is the ability to qualify who can send and receive event broadcasts.

The ability to ‘scope’ data using hierarchy is also of significant importance. By choosing a parent and scope you can specify where the data resides and qualify who can access it.

This figure shows the objects in the Stateflow machine hierarchy that can parent events and data.



Events and data have hierarchy and can be parented by a machine, a chart, or a state. Through the Explorer and the Event and Data properties dialog boxes, you can specify variations of parent and scope.

Where to Find More Information on Creating, Modifying, and Deleting Events and Data

For information on creating, modifying, and deleting event and data objects, see these sections: “Stateflow Explorer User Interface” on page 5-5, “Example: Getting Started Adding Events and Data” on page 5-16, and “Stateflow Finder User Interface” on page 5-20.

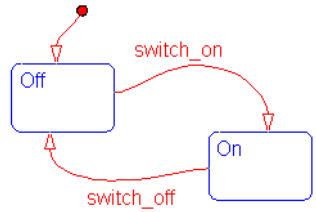
For information on data and events relevant to defining Stateflow interfaces, see Chapter 4, “Defining Stateflow Interfaces.”

Events

What Is an Event?

Events drive the Stateflow diagram execution. All events that affect the Stateflow diagram must be defined. Each event has a parent. The occurrence of an event causes the status of the states in the Stateflow diagram to be evaluated. The broadcast of an event can trigger a transition to occur and/or can trigger an action to be executed.

In this example, if state `Off` is active and the event `switch_on` is broadcast, the transition from `Off` to `On` is triggered. If state `On` is active and the event `switch_off` is broadcast, the transition from `On` to `Off` is triggered.



Event Properties

All events have these properties:

- Name (see “Event Name” on page 3-6)
- Description (see “Event Description” on page 3-7)
- Parent (see “Event Parent Types” on page 3-7, “Events, Hierarchy, and Scope” on page 3-9)
- Scope (see “Event Scope” on page 3-10, “Table of Event Parent and Scope Combinations” on page 3-15)
  - Port (if Scope is **Output to Simulink**) (see “Event Ports” on page 3-16)
  - Index (if Scope is **Input from Simulink**) (see “Event Indices” on page 3-16)
  - Trigger (if Scope is **Input from Simulink**) (see “Event Triggers” on page 3-17, “Determining Overall Input Trigger Type” on page 3-18)

See the section titled “Action Language” on page 6-42 for more information on the action language.

How Are Events Broadcast?

If events drive Stateflow diagram execution, an essential question is how are the events broadcast? Events are broadcast:

- Within the Stateflow diagram (**Local**) through event broadcast actions
- As input events from Simulink
- As **Imported** events from an external code source
- By the `send(event_name,state_name)` action language function

Events are broadcast in a top-down manner starting from the event's parent in the hierarchy. The `send` action language function allows refinement of the receive rights associated with the specified event. The send action language function sends the `event_name` specified to the `state_name` specified (the specified event is directed to be received by the state specified. See “Directed Event Broadcasting” on page 6-58 for more information on `send`.

Getting Started with Events

The flexibility and power provided by parenting and scoping events can be overwhelming and complex to begin with. The simplest case to consider is:

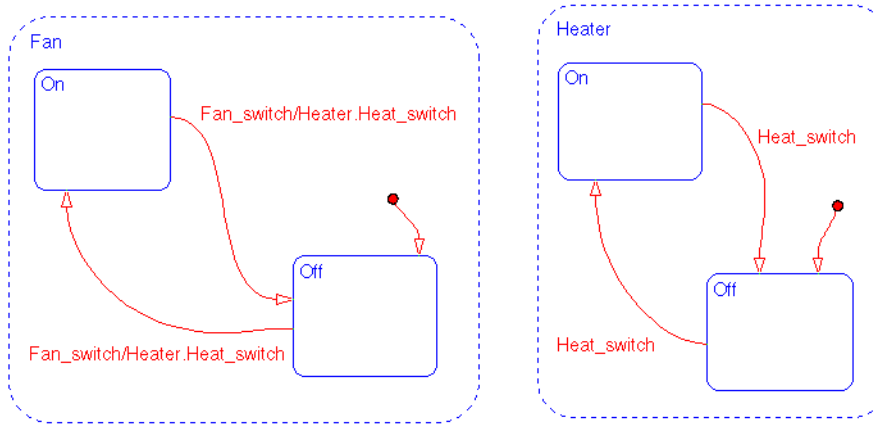
- Create a Simulink model containing a single Stateflow diagram
- Use the Simulink implicit event broadcast to awaken the Stateflow diagram
- Define events parented by the chart
- Define events scope defined as **Local**
- Specify **Local** event broadcasts within the Stateflow diagram

See “Example: Getting Started Adding Events and Data” on page 5-16 for an example of this simplest case.

Event Name

An event name can consist of any alphanumeric and special character combination, with the exception of embedded spaces. The name cannot begin with a numeric character. The length of a name is not restricted.

You can qualify an event name by prepending ancestry names. In this example, the transition action specifies a qualified event name `Heater.Heat_switch` to be broadcast.



Event Description

The description can consist of any ASCII text you want to enter to describe the object.

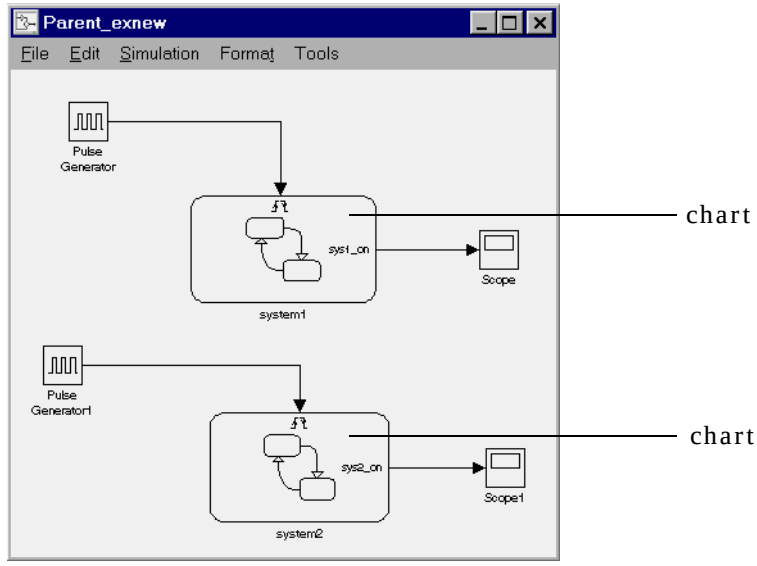
Event Parent Types

You must choose a parent when defining an event. You explicitly choose the parent through the Explorer. The chart is implicitly the parent if you use the

graphics editor **Add** menu. The parent and scope together define a range of access to events. Events can have parents with these object types:

- Machine

  The machine is the collection of all Stateflow blocks in the Simulink model. In this example, the machine is the collection of `system1` and `system2` Stateflow blocks.

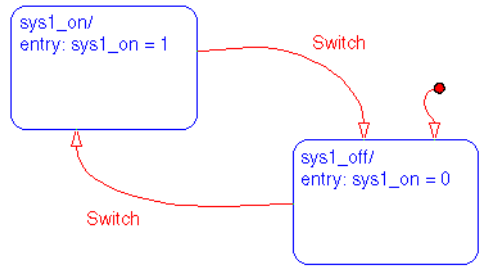


- Chart (Stateflow diagram)

  The chart (Stateflow diagram) refers to the specific Stateflow block. The Stateflow blocks titled `system1` and `system2` each represent an underlying chart that can parent objects within the Stateflow diagram.

- State

  The state refers to a specific state within the hierarchy. In the example, this is `system1`'s Stateflow diagram. Either of these states, `sys1_on` and `sys1_off` can be parents.



Events, Hierarchy, and Scope

Events have hierarchy (a parent) and scope. The parent and scope together define a range of access to events. Access to events is defined in terms of these two questions:

- Who can broadcast the event? (sender)

  The event's scope determines who can broadcast the event (has send rights). The event's parent has no effect on who can broadcast the event. By default, when you create a new event in the Explorer, the event scope is **Local**. A **Local** event can be broadcast from anywhere in the hierarchy of the machine. Changing the scope of the event to a different scope changes this behavior. See "Event Scope" on page 3-10 for the behavior associated with the scope value choices.

- Who can trigger on the event? (receiver)

  It is primarily the event's parent that determines who can trigger on the event (has receive rights). You must explicitly choose a parent when defining an event. By default, when you create a new event in the Explorer, the event's scope is **Local**. A **Local** event can trigger transitions from anywhere in the hierarchy of its parent. Changing the scope of the event from the default value of **Local** to one of the other values changes this default behavior. See "Event Scope" on page 3-10 for the behavior associated with the scope value choices. Appropriate selection of an event's parent allows you

to be specific about who can trigger on an event within the hierarchy. This directed event broadcasting eliminates unnecessary event processing.

Event Scope

An event's scope defines to what extent the object is known relative to the Stateflow block, the Simulink model, and sources external to the Simulink model. These are the options for the **Scope** property:

- **Local**
- **Input from Simulink**
- **Output to Simulink**
- **Exported**
- **Imported**

Local

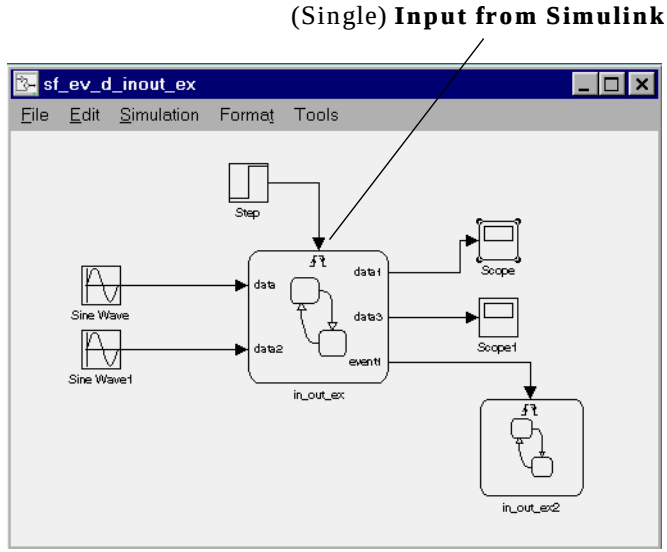
A **Local** event is broadcast and only received within the hierarchy of the parent. Valid parent types for a local event are machine, chart, and state.

Input from Simulink

An event that originates from the Simulink model and is an input to the Stateflow block is an **Input** event. **Input** events must be parented by the chart. **Input** events are broadcast to the chart and its offspring.

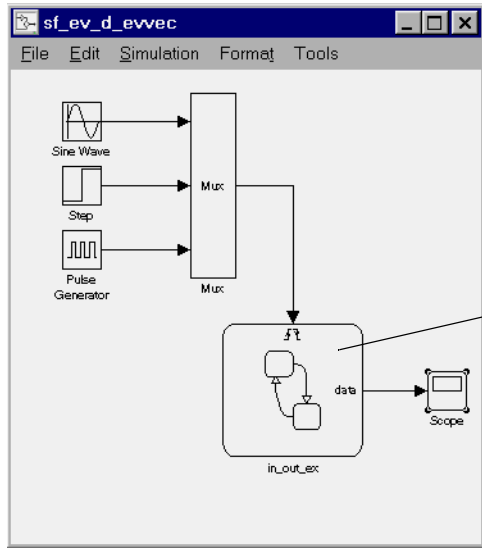
Events that are defined to be inputs are also interfaces to the Simulink model of which the Stateflow block is a part. When you define an event as an **Input from Simulink**, the appropriate notation is automatically added to the Stateflow block in the Simulink model. Event inputs are represented by the addition of a Simulink trigger port on the Stateflow block.

When you add an event **Input from Simulink**, a single Simulink trigger port is added to the Stateflow block.



Multiple event inputs from the Simulink model to the Stateflow block must be vectored into that single port using the Simulink MUX block. You must make

sure that the order of the input elements in the vector correspond properly to the index numbers defined in the event's property dialog box.



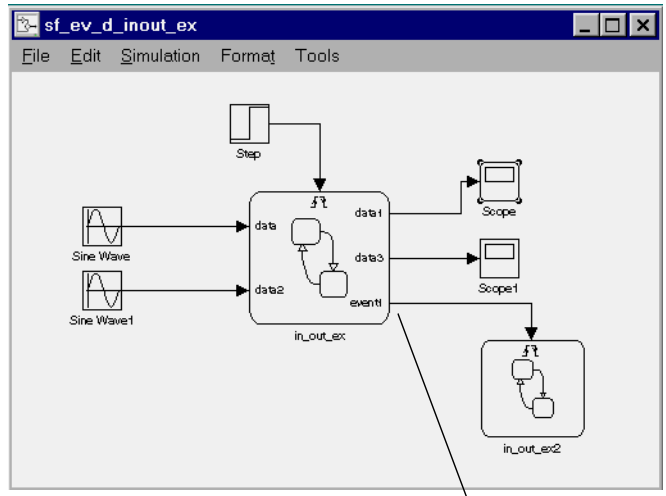
Three **Input from Simulink** events are defined (all **Either Edge** triggered)

Output to Simulink

An event that originates from the Stateflow block and is an output to its Simulink model is an **Output to Simulink** event. **Output to Simulink** events must be parented by the chart. **Output to Simulink** events are broadcast, using a port and connection, to a Simulink block or another Stateflow block.

Events that are defined to be outputs are also interfaces to the Simulink model of which the Stateflow block is a part. When you define an event to be an **Output to Simulink**, the appropriate notation is automatically added to the Stateflow block in the Simulink model. Each event output is represented by a

Simulink output port. Output events from the Stateflow block to the Simulink model are scalar.



Exported

Consider a real world example to clarify when to define an **Exported** event. You have purchased a communications pager. There are a few people you want to be able to page you, so you give those people your personal pager number. These people now know your pager number and can call that number and page you whatever you might be doing. You do not usually page yourself, but you can do so. Telling someone the pager number does not mean they have heard and recorded the number. It is the other person's responsibility to retain the number.

Similarly, you may want an external source (outside the Stateflow diagram, the machine, and the Simulink model) to be able to broadcast an event. By defining an event's scope to be **Exported**, that event is made available to external sources for broadcast purposes. **Exported** events must be parented by the machine because the machine is the (highest) level in the Stateflow hierarchy that can interface to external sources. The machine also retains the ability to broadcast the **Exported** event. Exporting the event does not imply anything about what the external source does with the information. It is the

responsibility of the external source to include the **Exported** event (in the manner appropriate to the source) to make use of the right to broadcast the event.

If the external source is another machine (most common scenario) then one machine defines an **Exported** event and the other machine defines the same event to be **Imported**. Stateflow generates the appropriate export and import event code for both machines. See “Exported Events” on page 4-25 for an example showing the imported event format required if an external code source is used.

Imported

Consider a real world example to clarify when to define an **Imported** event. Someone buys a communications pager and indicates you may want to use the pager number to page them in the future. They tell you the pager number and you take note of the number by writing it down. You can then use the number to page that person. You cannot receive the page because you do not own the pager.

Similarly, you may want to broadcast an event that is externally (outside the Stateflow diagram, the machine, and the Simulink model) defined. By defining an event's scope to be **Imported**, the event can be broadcast anywhere within the hierarchy of that machine (including any offspring of the machine). An **Imported** event's parent is external. However, the event needs an ‘adoptive’ parent to resolve symbols for code generation. An **Imported** event's adoptive parent must be the machine because the machine is the (highest) level in the Stateflow hierarchy that can interface to external sources. Importing the event does not imply anything about what the external source does with the information. It is the responsibility of the external source to make the **Imported** event available (in the manner appropriate to the source) to make use of the right to broadcast the event.

If the external source is another machine (most common scenario) then one machine defines as **Imported** event and the other machine defines the same event to be **Exported**. Stateflow generates the appropriate import and export event code for both machines. See “Imported Events” on page 4-27 for an example showing the exported event format required in the external code source.

Table of Event Parent and Scope Combinations

This table summarizes the answers to these access questions:

- Who can broadcast the event? (has send rights)
- Who can trigger on the event (has receive rights)?

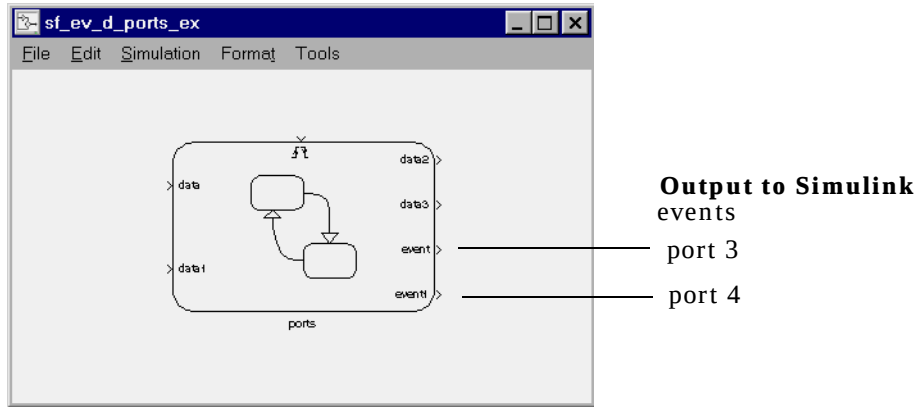
Offspring is defined as all substates of a parent.

Scope	Can be Parented by:	Send Rights	Receive Rights
Local	machine, chart, state	Any offspring of the machine	Parent and any offspring
Input from Simulink	chart	Simulink model and any offspring of the chart	Chart and all its offspring
Output to Simulink	chart	Simulink model and any offspring of the chart	Simulink model
Exported	machine	Any offspring of the machine and external sources	Any offspring of the machine
Imported	machine	Any offspring of the machine	External to the machine

The **Scope** determines send rights. The parent primarily determines receive rights. Events are broadcast in a top-down manner starting from the event's parent in the hierarchy. The send action language function allows refinement of the receive rights associated with the specified event. The send action language function sends the `event_name` specified to the `state_name` specified (the specified event is directed to be received by the state specified. See "Directed Event Broadcasting" on page 6-58 for more information on send.

Event Ports

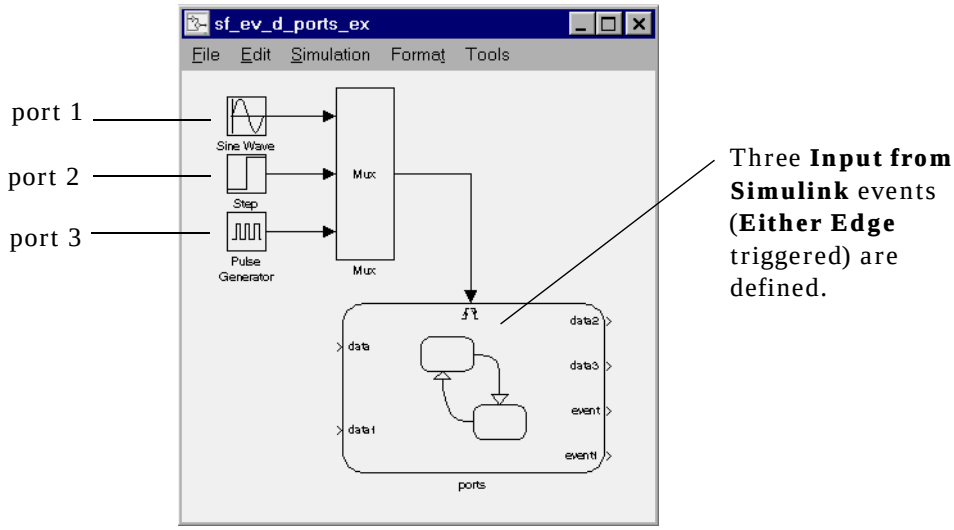
Ports are used to connect the Stateflow block to other Simulink blocks. Each event output in a Stateflow diagram corresponds to a port on the Stateflow block. Ports have numbers that indicate the relative position of that port on the Stateflow block. Each event output is represented by the addition of a Simulink port on the right side of the Stateflow block. Event ports appear below any data ports on the right side of the Stateflow block. Each port on the Stateflow block must be connected to a port on an appropriate block in the Simulink model.



Event Indices

Event indices are used to correlate the order of the event defined as an input from Simulink within the Stateflow diagram with the vector of inputs from Simulink. Event inputs are represented by the addition of a Simulink trigger port on the top of the Stateflow block. If there are multiple event inputs they are vectored into that single trigger port using a Simulink MUX block. You

must make sure that the order of the input elements in the vector corresponds properly to the index numbers defined in the **Event property** dialog box.



Event Triggers

A trigger defines how a Stateflow block's input and output events are handled in the context of their Simulink model. The **Trigger** type indicates what kind of signal has meaning for the input or output event. **Trigger** types fall into two general categories: edge or function call triggered. Each Stateflow block can have only one overall **Trigger** type.

The **Trigger** property can have these values:

- **Either Edge**

  Either a rising or a falling trigger, where the control signal changes either from 0 to a nonzero value, from a positive to a negative value, or from a negative to a positive value.
- **Rising Edge**

  A rising trigger, where the control signal changes either from 0 to a positive value or from a negative to a positive value.
- **Falling Edge**

  A falling trigger, where the control signal changes either from 0 to a negative value or from a positive to a negative value.
- **Function Call**

  A function call trigger operates essentially like a programming subroutine call. When the system executes the step where the function call is specified, the triggered subsystem executes and then returns to the next statement in the execution sequence. Using function call triggers, the Stateflow block can control the execution of other Simulink blocks in the model.

See “Defining the Stateflow Block Update Method” on page 4-4 for more information.

Determining Overall Input Trigger Type

Each Stateflow block can only have one overall **Trigger** type. If multiple input events are defined for a Stateflow block, the following algorithm is applied to arrive at a single **Trigger** type for the Stateflow block:

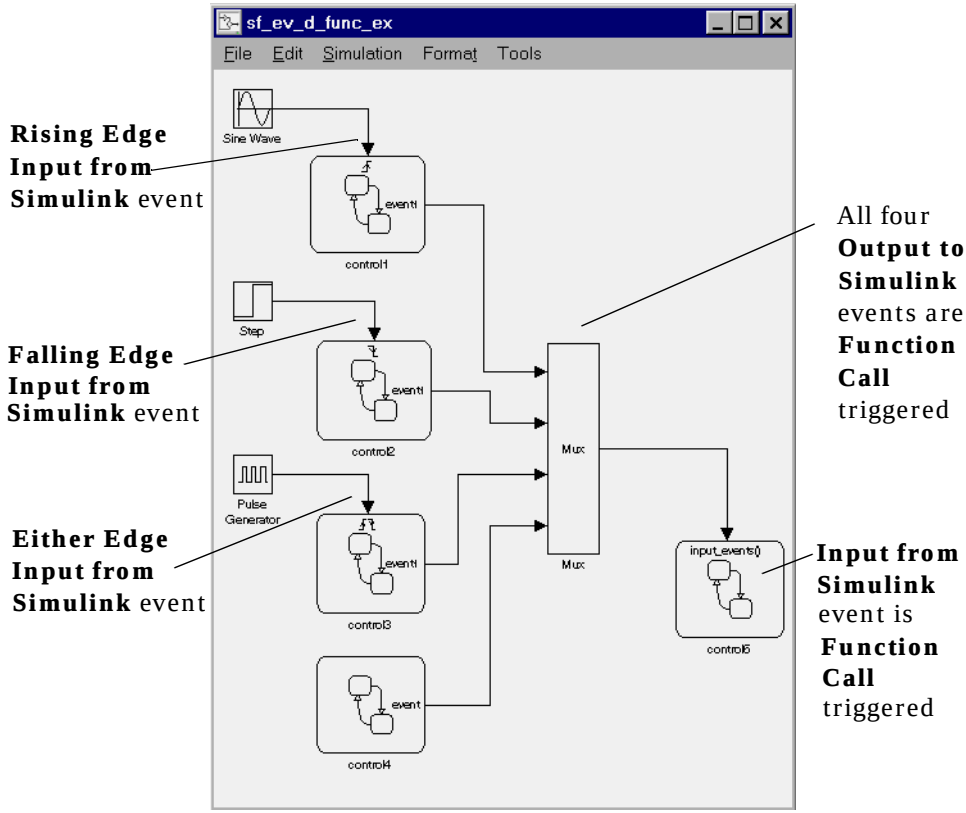
- **Function Call** triggers take precedence over **Edge** trigger types. If one of the **Input from Simulink** event's trigger type is **Function Call**, then the Stateflow block **Trigger** type is set to **Function Call**. See “Combining Function Call and Edge Triggers” on page 3-19 for an example of how to effectively model a combination of **Function Call** and **Edge** trigger types.
- If all events are **Rising Edge** trigger type, then the Stateflow block **Trigger** type is set to **Rising Edge.**
- If all events are **Falling Edge** trigger type, then the Stateflow block **Trigger** type is set to **Falling Edge**.
- Otherwise, the Stateflow block **Trigger** type is set to **Either Edge.**

The **Trigger** type must be consistent between the source and destination of Simulink blocks. For example, if a Stateflow block has a **Function Call Trigger** type and is connected to a Simulink triggered subsystem, the triggered subsystem must define **function-call** triggering.

See “Defining the Stateflow Block Update Method” on page 4-4 for more information.

Combining Function Call and Edge Triggers

Since **Function Call** triggers take precedence over **Edge** trigger types, any Edge triggered events defined in the input vector will be ignored. This example illustrates an alternative method to combine **Function Call** triggered events and **Edge** triggered events.



Implicit Local Events

Overview

There is a set of implicit local events that can be used within the Stateflow diagram. These events are very useful to define conditional activity based on when a specified state has been entered or exited or when a data value has changed. Only implicit events that are referenced in the Stateflow diagram are broadcast.

Implicit local events are parented by the chart. Any offspring of the chart can trigger on the event (receive rights). The state_name and data_object_name must be unique within the chart. The names can be qualified with ancestor names in the hierarchy if necessary.

Types of Implicit Events

You can specify these events in any Stateflow diagram without performing any explicit definition. The set includes:

- `entry(state_name);`

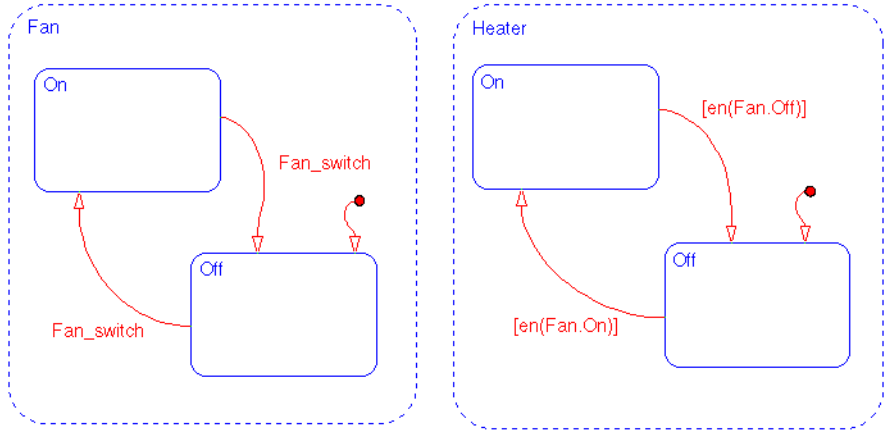
  Generates a local event when the specified state is entered. The shorthand notation is `en(state_name)`.
- `exit(state_name);`

  Generates a local event when the specified state is exited. The shorthand notation is `ex(state_name)`.
- `change(data_object_name);`

  Generates a local event when there is a change to the numeric value of the specified data object name. The shorthand notation is `chg(data_object_name)`.

Example: Implicit Local Event

This example uses the implicit local event `entry(state_name)`.



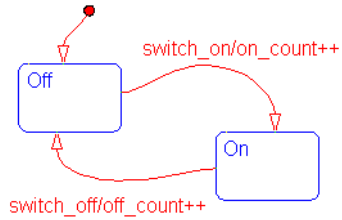
`Fan` and `Heater` are parallel (AND) superstates. By default, the first time the Stateflow diagram is awakened by an event, the states `Fan.Off` and `Heater.Off` become active. The first time event `Fan_switch` occurs, the transition from `Fan.Off` to `Fan.On` occurs. When `Fan.On`'s entry action executes, an implicit local event is broadcast (ie., `en(Fan.On) == 1`). This event broadcast triggers the transition from `Heater.Off` to `Heater.On` (guarded by the condition `[en(Fan.On)]`. Similarly, when the system transitions from `Fan.On` to `Fan.Off` and the implicit local event `Fan.Off` is broadcast, the transition from `Heater.On` to `Heater.Off` is triggered (guarded by the condition `[en(Fan.Off)]`.

Data

What Is a Data Object?

Data objects are numerical values in the Stateflow diagram. Each data object has a parent. Data is often manipulated in the actions that can be associated with states and/or transitions.

For example, if state `Off` is active and the event `switch_on` is broadcast, the transition from `Off` to `On` is triggered. The data `on_count` is incremented once the destination state, `On`, has been determined. If state `On` is active and the event `switch_off` is broadcast, the transition from `On` to `Off` is triggered. The data `off_count` is incremented once state `Off` has been determined as the destination.



The value of data is determined either from within the Stateflow diagram through actions or from outside of the Stateflow diagram, either through Simulink or from an external source.

All data objects have these properties:

- Name (see “Data Name” on page 3-24)
- Description (see “Data Description” on page 3-24)
- Parent (see “Data Parent Types” on page 3-25, “Data, Hierarchy, and Scope” on page 3-26)
- Scope (see “Data Scope” on page 3-27, “Table of Data Scope and Parent Combinations” on page 3-32)
  - Port (if Scope is **Input from Simulink** or **Output to Simulink**) (see “Data Ports” on page 3-32)
- Data Attributes
  - Units (See “Units” on page 3-33)
  - Type (See “Type” on page 3-34)
  - Initial, Minimum, and Maximum (See “Initial Value” on page 3-34, “Minimum Value” on page 3-34, and “Maximum Value” on page 3-34)

See the section titled “Action Language” on page 6-42 for more information on the action language.

Getting Started with Data

The flexibility and power provided by parenting and scoping data can be overwhelming and complex to begin with. The simplest case to consider to start with is:

- Create a Simulink model containing a single Stateflow diagram
- Use the Simulink implicit event broadcast to awaken the Stateflow diagram
- Define data parented by the chart
- Define the data scope to be **Local**
- Specify **Local** events broadcast in the Stateflow diagram

See “Example: Getting Started Adding Events and Data” on page 5-16 for an example of this simplest case.

Data Name

A data name can consist of any alphanumeric and special character combination, with the exception of embedded spaces. The name cannot begin with a numeric character. The length of a name is not restricted.

Data Description

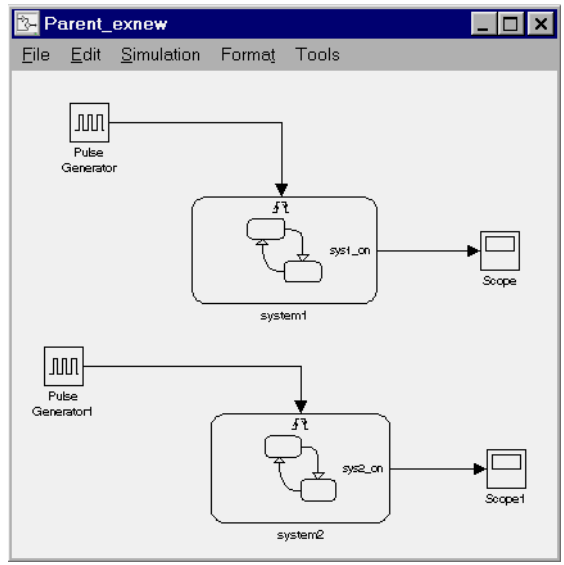
The description can consist of any ASCII text you want to enter to describe the object.

Data Parent Types

You must choose a parent when defining a data object. You explicitly choose the parent through the Explorer. The chart is implicitly chosen as the parent if you use the graphics editor **Add** menu. Data objects can have parents with these object types:

- Machine

  The machine is the collection of all Stateflow blocks in the Simulink model. In this example, the machine is the collection of `system1` and `system2` Stateflow blocks.

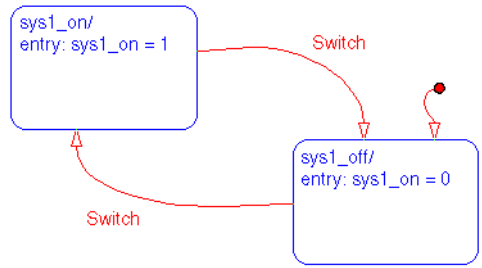


- Chart

  The chart (Stateflow diagram) refers to the specific Stateflow block. The Stateflow blocks titled `system1` and `system2` each represent an underlying chart that can parent objects within the Stateflow diagram.

- State

  The state refers to a specific state within the hierarchy. In the example, this is `system1`'s Stateflow diagram. Either of these states, `sys1_on` and `sys1_off` can be parents.



Data, Hierarchy, and Scope

Data objects have hierarchy (a parent) and scope. Hierarchy and scope together define a range of access to data objects. A data object is accessible only within its parent (including siblings).

Access to data is defined in terms of these two questions:

- Where does the data reside?

  The data object's parent determines where the data resides.

- Who accesses the data?

  The data object's parent and scope determine who can access the data. By default the data's **Scope** value is **Local**. **Local** data can be accessed from anywhere in the hierarchy of its parent. Changing the **Scope** of the data from the default value of **Local** to one of the other values changes this default behavior. See "Data Scope" on page 3-27 for the behavior associated with the **Scope** value choices.

Data Scope

A data object's scope defines to what extent the object is known relative to the Stateflow block, the Simulink model, and sources external to the Simulink model. These are the options for the **Scope** property:

- **Local**
- **Input from Simulink**
- **Output to Simulink**
- **Temporary**
- **Workspace**
- **Constant**
- **Exported**
- **Imported**

Local

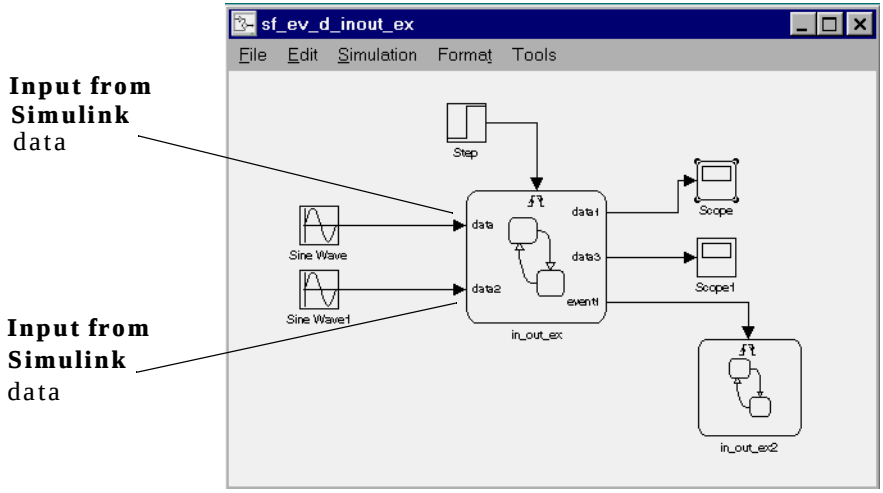
A **Local** data object resides and is accessible only within the hierarchy of the parent. Valid **Parent** types are machine, chart, and state.

Input from Simulink

A data object that originates from the Simulink model and is an input to the Stateflow block is an **Input from Simulink** data. **Input from Simulink** data must be parented by the chart. **Input from Simulink** data objects are accessible by the chart and its offspring.

Data defined to be an input is also an interface to the Simulink model of which the Stateflow block is a part. When you define a data object as an **Input from Simulink**, the appropriate notation is automatically added to the Stateflow block in the Simulink model. Each data **Input from Simulink** is represented

by a Simulink input port. Multiple data inputs to the Stateflow block must be scalar (they cannot be vectorized).

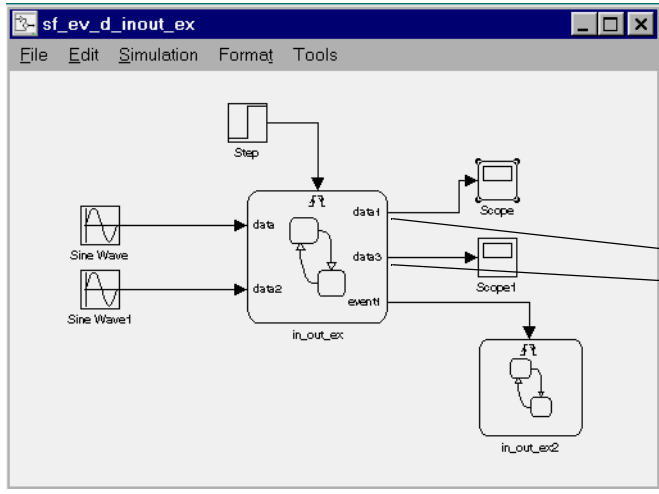


Output to Simulink

A data object that originates from the Stateflow block and is an output to its Simulink model is an **Output to Simulink** data object. **Output to Simulink** data must be parented by the chart. **Output to Simulink** data objects are passed, using a port and connection, to a Simulink block or another Stateflow block.

Data defined to be an **Output to Simulink** is also an interface to the Simulink model of which the Stateflow block is a part. When you define a data object as an **Output to Simulink**, the appropriate notation is automatically added to the Stateflow block in the Simulink model. Each data **Output to Simulink** is represented by a Simulink output port. Multiple data outputs from the

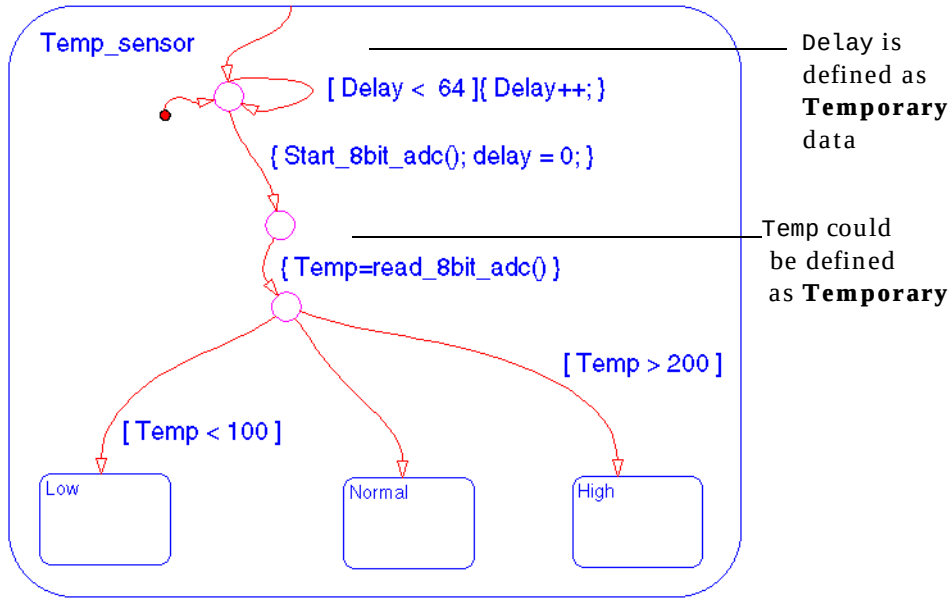
Stateflow block to the Simulink model must be scalar (they cannot be vectorized).



Temporary

Temporary data is non-persistent. **Temporary** data is accessible only by its parent. Defining a loop counter to be **Temporary** is a good use of this **Scope** since the value is used only as a counter and the value does not need to persist. The **Temporary** data remains defined and accessible only while the parent is executing. For a state, the **Temporary** data is defined when that state is active and processing state actions. For a chart, the **Temporary** data is defined and accessible when the chart is active and processing condition and/or transition

actions. This is an example Stateflow diagram of an Analog to Digital converter that uses **Temporary** data.



`Delay` is used only as a counter in the loop. Once the delay loop is complete, the value of `Delay` is no longer needed. If you were interested only in whether the `Low`, `Normal`, or `High` state is active, the value of `Temp` could be defined as **Temporary** as well.

Workspace

A **Workspace** data object is a value that typically does not change during a simulation. Workspace data objects are passed in from MATLAB and/or Simulink. Workspace data objects can be defined at the MATLAB command line or through Simulink mask initialization. Stateflow will attempt to resolve the symbol first locally, then in Simulink, and lastly in MATLAB. See the *Using Simulink* document for more information on mask initialization.

Constant

A **Constant** data object is read-only and retains the initial value set in its data properties dialog box.

Exported

An **Exported** data object is parented by and declared within the machine.

Exporting a data object provides the ability for external sources to access the data object; the machine also retains the ability to access the data object.

It is the responsibility of the external source to include the **Exported** data (in the manner appropriate to the source) to make use of these rights.

If the external source is another machine (most common scenario) then one machine defines an **Exported** data object and the other machine defines the same data object to be **Imported**. Stateflow generates the appropriate export and import data code for both machines. See “Exported Data” on page 4-28 for an example showing the imported data format required if an external code source is used.

Imported

An **Imported** data object is defined external to the machine; the parent is consequently external to the machine.

Importing a data object provides access to the data object.

It is the responsibility of the external source to make the **Imported** data available (in the manner appropriate to the source) to make use of these rights.

If the external source is another machine (most common scenario) then one machine defines an **Imported** data object and the other machine defines the same data object to be **Exported**. Stateflow generates the appropriate import and export data code for both machines. See “Imported Data” on page 4-29 for an example showing the exported data format required if an external code source is used.

Table of Data Scope and Parent Combinations

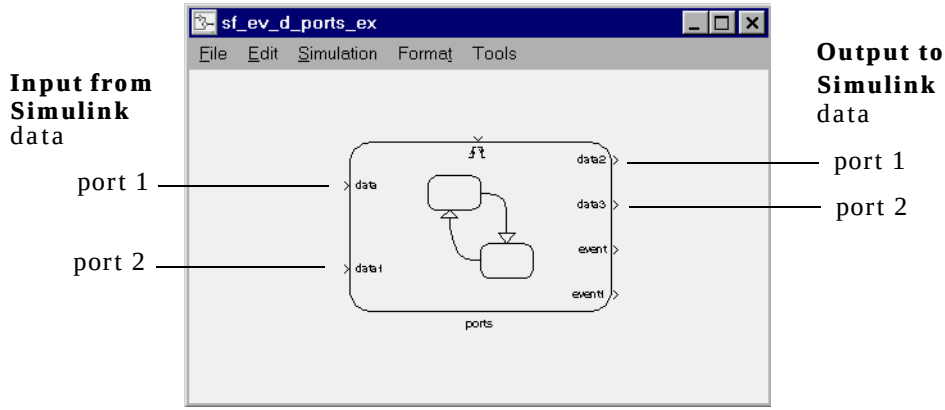
This table summarizes who can access the data. Offspring is defined as all substates of a parent.

Scope	Can be Parented by:	Access Rights
Local	machine, chart, state	Parent and any of its offspring.
Input from Simulink	chart	Simulink and any offspring of the chart.
Output to Simulink	chart	Simulink and any offspring of the chart.
Temporary	chart, state	Parent only (none of its offspring).
Workspace	machine, chart	Parent and any of its offspring.
Constant	machine, chart, state	Parent and any of its offspring.
Exported	machine	Any offspring of the machine and external sources.
Imported	machine	Any offspring of the machine.

Data Ports

Ports are used to connect the Stateflow block to other Simulink blocks. Each input or output in a Stateflow diagram corresponds to a port on the Stateflow block. Ports have numbers that indicate the relative position of that port on the Stateflow block. Each data input is represented by the addition of a Simulink port on the left side of the Stateflow block. Each data output is represented by the addition of a Simulink port on the right side of the Stateflow block. Each

port on the Stateflow block must be connected to a port on an appropriate block in the Simulink model.



Data Attributes

Data objects have these data type attributes that define how data objects are represented in the generated code.

- Units
- Type
- Initial, Minimum, and Maximum

Units

Units is a user-entered string that describes the units of the data type.

Type

These data types are supported:

- **Real (double)**
- **Real (float)**
- **Integer (long)**
- **Integer (short)**
- **Boolean (unsigned char)**
- **Word (unsigned long)**
- **Word (unsigned short)**
- **Byte (unsigned char)**

Initial Value

Scalar initial value of the data object. (0 by default)

Minimum Value

Smallest scalar value of the data object. (-inf (negative infinity) by default)

Maximum Value

Largest scalar value of the data object. (inf (positive infinity) by default)

4

Defining Stateflow Interfaces

Overview 4-2

Defining the Stateflow Block Update Method 4-4

Defining Output to Simulink Event Triggers 4-10

Event Inputs from Simulink 4-16

Data Inputs from Simulink 4-18

Event Outputs to Simulink 4-20

Data Outputs to Simulink 4-22

MATLAB Workspace 4-24

Defining the Interface to External Sources 4-25

Overview

Interfaces to Stateflow

Each Stateflow block interfaces to its Simulink model. Each Stateflow block may interface to sources external to the Simulink model (data, events, custom code). Events and data are the Stateflow objects that define the interface from the Stateflow block's point of view.

Events can be local to the Stateflow block or can be propagated to and from Simulink and sources external to Simulink. Data can be local to the Stateflow block or can be shared with and passed to the Simulink model and to sources external to the Simulink model.

The Stateflow block interface includes:

- Physical connections between Simulink blocks and the Stateflow block
- Event and data information exchanged between the Stateflow block and either its Simulink model or external sources
- MATLAB workspace
- Definitions in external code sources

Typical Tasks to Define Stateflow Interfaces

Defining the interface for a Stateflow block can involve some or all of these tasks:

- Defining the Stateflow block update method
- Defining **Output to Simulink** events
- Adding and defining nonlocal events and nonlocal data within the Stateflow diagram
- Defining relationships with any external sources

The tasks are presented in this section in the order of appearance in this list. This could be a typical sequence. You may find a particular sequence complements your model development process better than another.

Where to Find More Information on Events and Data

See these sections for conceptual information on data and events: “Events” on page 3-5 and “Data” on page 3-22. These references in particular are relevant to defining the interface:

- “Event Parent Types” on page 3-7
- “Events, Hierarchy, and Scope” on page 3-9
- “Event Scope” on page 3-10
- “Table of Event Parent and Scope Combinations” on page 3-15
- “Event Ports” on page 3-16
- “Event Triggers” on page 3-17
- “Data Parent Types” on page 3-25
- “Data, Hierarchy, and Scope” on page 3-26
- “Data Scope” on page 3-27
- “Table of Data Scope and Parent Combinations” on page 3-32
- “Data Ports” on page 3-32
- “Data Attributes” on page 3-33

For information on creating, modifying, and deleting event and data objects, see these sections: “Stateflow Explorer User Interface” on page 5-5, “Example: Getting Started Adding Events and Data” on page 5-16, and “Stateflow Finder User Interface” on page 5-20.

Defining the Stateflow Block Update Method

Stateflow Block Update Methods

Stateflow blocks are Simulink subsystems. You have some flexibility in defining the type of Simulink subsystem of a particular Stateflow block. The chart is awakened when an event occurs. You can choose from these methods of having the chart awakened, entered, and executed:

- Triggered/Inherited

  This is the default update method.

  - Triggered

    The Stateflow block is explicitly triggered by a signal originating from a connected Simulink block. The edge trigger can be set to **Rising**, **Falling**, **Either**, or **Function Call**.

  - Inherited

    The Stateflow block inherits (implicitly) triggers from the Simulink model. These implicit events are the sample times (discrete-time or continuous) of the Simulink signals providing inputs to the chart. The sample times are determined by Simulink to be consistent with various rates of all the incoming signals.

- Sampled

  Simulink will awaken (sample) the Stateflow block at the rate you specify. An implicit event is generated by Simulink at regular time intervals corresponding to the specified rate. The sample time is in the same units as the Simulink simulation time. Note that other blocks in the Simulink model may have different sample times.

- Continuous

  Simulink will awaken (sample) the Stateflow block at each step in the simulation, as well as at intermediate time points that may be requested by the Simulink solver. This method is consistent with the continuous method in Simulink.

See *Using Simulink* for more information on these types of Simulink subsystems.

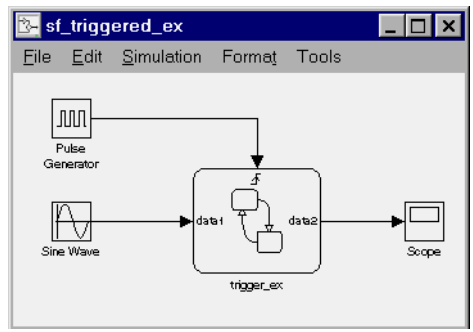
Defining a Triggered Stateflow Block

These are essential conditions that define an edge-triggered Stateflow block:

- The chart **Update method** (set in the **Chart Properties** dialog box) is set to **Triggered or Inherited**. (See "Editing Chart (Stateflow Diagram) Properties" on page 2-19)
- The chart has an **Input from Simulink** event defined and an edge-trigger type specified. Events are added and defined using either the graphics editor **Add** menu or the Explorer. See "Add Menu" on page 2-48 or "Tasks to Define an Event Input from Simulink" on page 4-16)

Example: Triggered Stateflow Block

A Pulse Generator block connected to the trigger port of the Stateflow block is an example of an edge-triggered Stateflow block. The **Input from Simulink** event has a **Rising Edge** trigger type.



If more than one **Input from Simulink** event is defined, the sample times are determined by Simulink to be consistent with various rates of all the incoming signals. The outputs of a Triggered Stateflow block are held after the execution of the block.

Defining a Sampled Stateflow Block

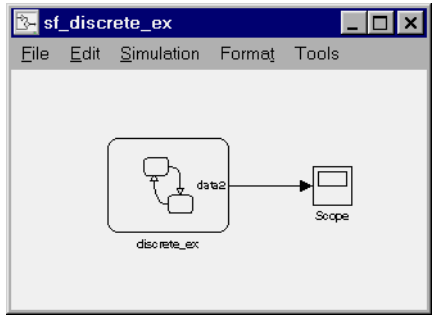
There are two ways you can define a sampled Stateflow block. Setting the chart **Update method** (set in the **Chart Properties** dialog box) to **Sampled** and entering a **Sample Time** value defines a sampled Stateflow block. (See "Editing Chart (Stateflow Diagram) Properties" on page 2-19.)

Alternatively, you can add and define an **Input from Simulink** data object. Data is added and defined using either the graphics editor **Add** menu or the Explorer. (See "Add Menu" on page 2-48 or "Tasks to Define a Data Input from Simulink" on page 4-18) The chart sample time is determined by Simulink to be consistent with the rate of the incoming data signal.

The **Sample Time** (set in the **Chart Properties** dialog box) takes precedence over the sample time of any **Input from Simulink** data.

Example: Sampled Stateflow Block

A Stateflow block that is not explicitly triggered via the trigger port can be triggered by Simulink by specifying a discrete sample rate. You can specify a **Sample Time** in the Stateflow diagram's **Chart properties** dialog box. The Stateflow block is then called by Simulink at the defined, regular sample times.



The outputs of a sampled Stateflow block are held after the execution of the block.

Defining an Inherited Stateflow Block

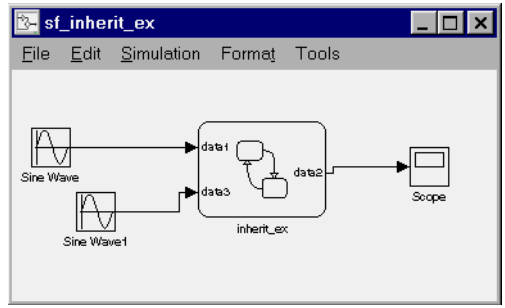
These are essential conditions that define an inherited trigger Stateflow block:

- The chart **Update method** (set in the **Chart Properties** dialog box) is set to **Triggered or Inherited**. (See "Editing Chart (Stateflow Diagram) Properties" on page 2-19)
- The chart has an **Input from Simulink** data object defined (added and defined using either the graphics editor **Add** menu or the Explorer). (See "Add Menu" on page 2-48 or "Tasks to Define a Data Input from Simulink" on page 4-18) The chart sample time is determined by Simulink to be consistent with the rate of the incoming data signal.

Example: Inherited Stateflow Block

A Stateflow block that is not explicitly triggered via the trigger port nor is a discrete sample time specified can be triggered by Simulink. The Stateflow block is called by Simulink at a sample time determined by Simulink.

In this example, more than one Input from Simulink data object is defined. The sample times are determined by Simulink to be consistent with the rates of both incoming signals.



The outputs of an inherited trigger Stateflow block are held after the execution of the block.

Defining a Continuous Stateflow Block

To define a continuous Stateflow block, the chart **Update method** (set in the **Chart Properties** dialog box) is set to **Continuous**. (See “Editing Chart (Stateflow Diagram) Properties” on page 2-19)

Considerations in Choosing Continuous Update

The availability of intermediate data makes it possible for the solver to ‘back up’ in time to precisely locate a ‘zero crossing’. Refer to *Using Simulink* for further information on zero crossings. Use of the intermediate time point information can provide increased simulation accuracy.

To support the **Continuous** update method, Stateflow keeps an extra copy of all its data.

In most cases, including continuous-time simulations, the **Inherited** method provides consistent results. The timing of state and output changes of the Stateflow block is entirely consistent with that of the continuous plant model.

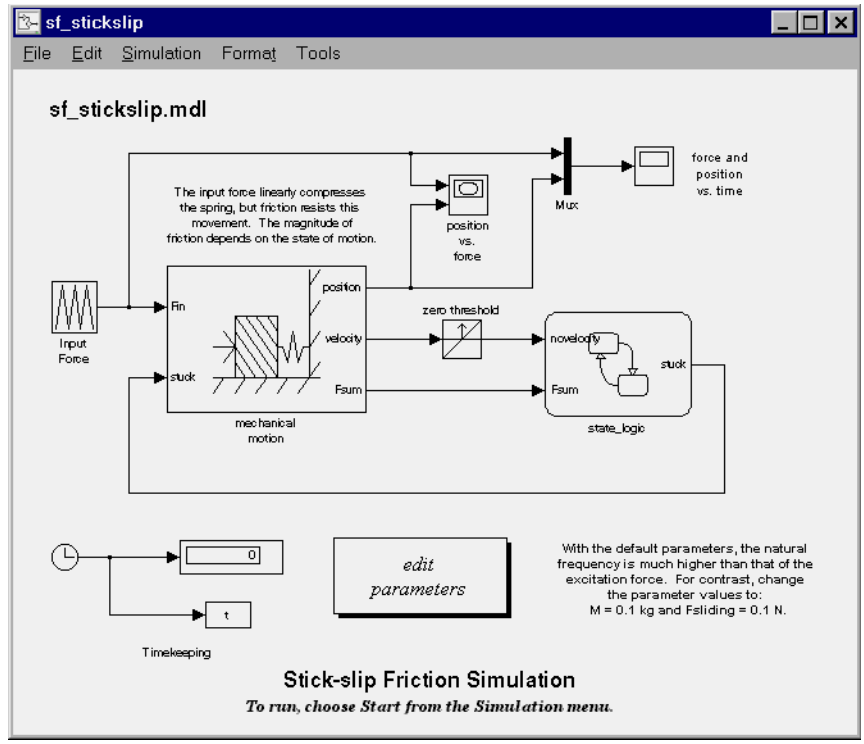
There are situations when changes within the Stateflow block must be felt immediately by the plant and a **Continuous** update is needed:

- Data **Output to Simulink** that is a direct function of data **Input from Simulink** and the data is updated by the Stateflow diagram (state `during` actions in particular).
- Models in which Stateflow states correspond to intrinsic physical states such as the onset of static friction or the polarity of a magnetic domain. This is in contrast to states that are assigned, for example, as modes of control strategy.

Example: Continuous Stateflow Block

Simulink will awaken (sample) the Stateflow block at each step in the simulation, as well as at intermediate time points that may be requested by the Simulink solver. This method is consistent with the continuous method in Simulink.

In this example (provided in the Examples/Stick Slip Friction Demonstration block), the chart **Update method** (set in the **Chart Properties** dialog box) is set to **Continuous**.



Defining Output to Simulink Event Triggers

Overview

Stateflow block output events connect to other Simulink blocks or Stateflow blocks. There are two main options for trigger type:

- Edge-triggered
- Function call

Simulink controls the execution of edge-triggered subsystems. The function call mechanism is a means by which Stateflow executes a subsystem essentially outside of Simulink's direct control. Use a function call trigger to have the Stateflow block control the execution of the connected Simulink block. Function call subsystems are never executed directly by Simulink.

See these examples for more information:

- "Example: Using Function Call Output Events" on page 4-11
- "Example: Function Call Semantics" on page 4-12
- "Example: Edge-Triggered Semantics" on page 4-13

Defining Function Call Output Events

These are essential conditions that define the use of function call output events:

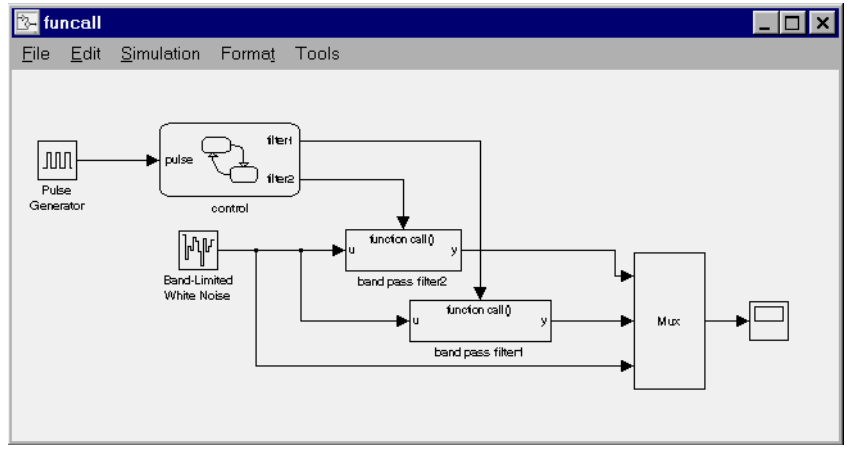
- The chart has an **Output to Simulink** event with a **Function Call** trigger type defined (added and defined using either the graphics editor **Add** menu or the Explorer. See "Add Menu" on page 2-48 or "Stateflow Explorer User Interface" on page 5-5 for more information.)
- The Simulink block connected to the function call event **Output to Simulink** has the **Trigger** type field set to **function-call**.
- Stateflow blocks that have feedback loops from a block triggered by a function call should avoid placing any other blocks in the connection lines between the two blocks.

Example: Using Function Call Output Events

A function call trigger operates essentially like a programming subroutine call. When the system executes the step where the function call is specified, the triggered subsystem executes and then returns to the next statement in the execution sequence. Using function call triggers, the Stateflow block can control the execution of other Simulink blocks in the model.

Use a function call event output when you want a Stateflow block (logic portion/control flow) to control one or more Simulink blocks (algorithmic portion/data flow).

This example shows a use of function call output events.

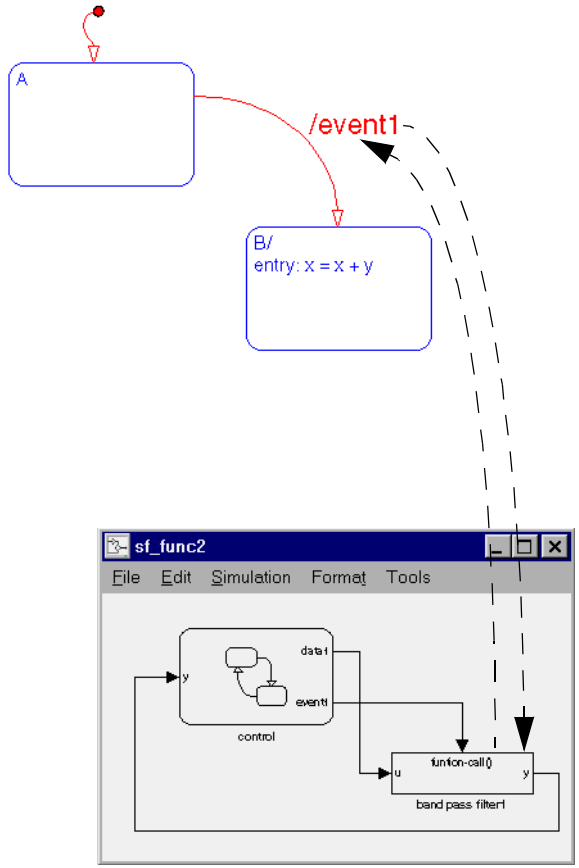


The `control` block is a Stateflow block that has one data input called `pulse` and two event **Function Call** outputs called `filter1` and `filter2`. A pulse generator provides input data to the `control` block. Within the `control` block, a determination is made whether to make a function call to `filter1` or `filter2`. If, for example, the **Output to Simulink** event **Function Call** `filter1` is broadcast, the band pass `filter1` block executes and then returns to the next execution step in the `control` block. As part of its execution, band pass `filter1` receives unfiltered input data and outputs filtered data for display on a scope.

The Stateflow block controls the execution of band pass `filter1` and band pass `filter2`.

Example: Function Call Semantics

In this example the transition from state `A` to state `B` (in the Stateflow diagram) has a transition action that specifies the broadcast of `event1`. `event1` is defined in Stateflow to be an **Output to Simulink** with a **Function Call** trigger type.The Stateflow block output port for `event1` is connected to the trigger port of the `band pass filter1` Simulink block. The `band pass filter1` block has its **Trigger** type field set to **function-call**.



This sequence is followed when state A is active and the transition from state A to state B is valid and is taken:

1. State A exit actions execute and complete.
2. State A is marked inactive.
3. The transition action is executed and completed. In this case the transition action is a broadcast of `event1`. Because `event1` is an event **Output to Simulink** with a function call trigger, the band pass `filter1` block executes and completes, and then returns to the next statement in the execution sequence. The value of y is fed back to the Stateflow diagram.
4. State B is marked active.
5. State B entry actions execute and complete (`x = x + y`). The value of y is the updated value from the band pass `filter1` block.
6. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

Defining Edge-Triggered Output Events

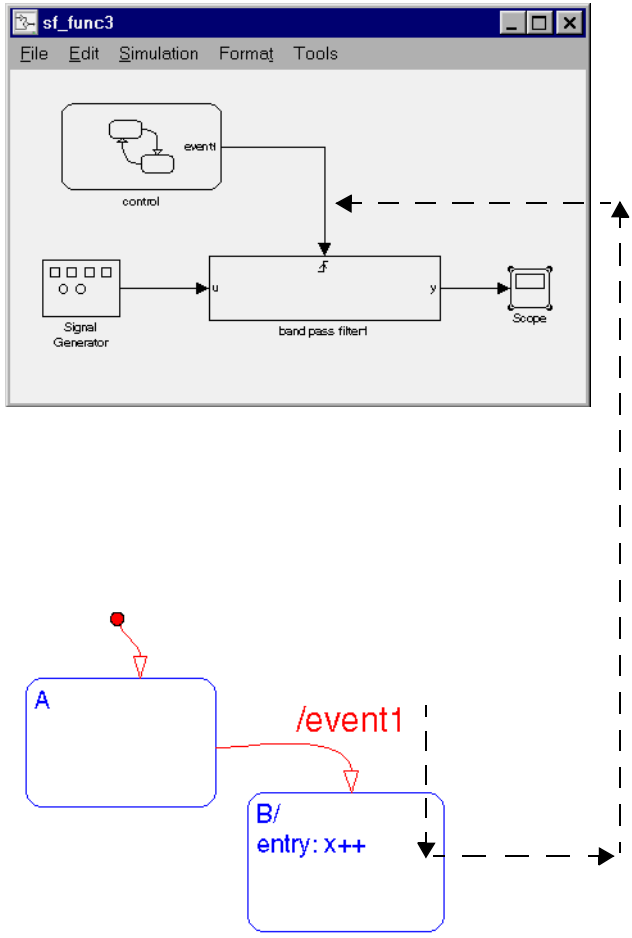
These are essential conditions that define the use of triggered output events:

- The chart has an **Output to Simulink** event with one of these edge-triggered types defined: **Either Edge**, **Rising Edge**, **Falling Edge**. (Events are added and defined using either the graphics editor **Add** menu or the Explorer. See “Add Menu” on page 2-48 or “Stateflow Explorer User Interface” on page 5-5 for more information.)
- The Simulink block connected to the edge triggered event **Output to Simulink** has the **Trigger** type field set to the equivalent edge triggering type.

Example: Edge-Triggered Semantics

In this example the transition from state A to state B (in the Stateflow diagram) has a transition action that specifies the broadcast of `event1`. `event1` is defined in Stateflow to be an **Output to Simulink** with a **Rising edge** trigger type. The Stateflow block output port for `event1` is connected to the trigger port of the

`band pass filter1` Simulink block. The `band pass filter1` block has its **Trigger type** field set to **rising**.



This sequence is followed when state A is active and the transition from state A to state B is valid and is taken:

1. State A exit actions execute and complete.
2. State A is marked inactive.
3. The transition action, an edge triggered **Output to Simulink** event, is registered (but not executed). Simulink is controlling the execution and execution control does not shift until the Stateflow block completes.
4. State B is marked active.
5. State B entry actions execute and complete (`x = x++`).
6. The Stateflow diagram goes back to sleep waiting to be awakened by another event.
7. The band pass `filter1` block is triggered, executes, and completes.

Event Inputs from Simulink

Tasks to Define an Event Input from Simulink

These tasks describe how to add and define the necessary fields for an event input from Simulink:

- Add an event choosing a chart as the parent of the event
- Choose **Input from Simulink** as the **Scope**
- Specify the **Trigger** type
- Apply and save the changes

Add an Event Choosing a Chart as the Parent

These steps describe how to add an event:

1. Choose **Explore** from the graphics editor **Tools** menu to invoke the Explorer.
2. Select the chart object in the hierarchy that you want to be the event's parent.

   You must explicitly choose a parent to create an event. Choosing the chart to be the parent of the event enables receive rights to Simulink, to the chart, and all its offspring.
3. Choose **Event** from the Explorer **Add** menu. The **Event properties** dialog box is displayed.
4. Enter a name in the **Name** field.

Choose Input from Simulink as the Scope

Once you have chosen the chart as the parent, the choice of valid scopes includes: **Local**, **Input from Simulink**, or **Output to Simulink**.

Choose **Input from Simulink** as the **Scope** to enable send rights to Simulink and any offspring of the chart and to enable receive rights to the chart and all of its offspring.

When you add an event input, a single Simulink trigger port is added to the top of the Stateflow block.

Select the Trigger

The trigger defines how the Stateflow block's input events are handled in the context of their Simulink model. The **Trigger** type indicates what kind of signal has meaning for the input event. The **Trigger** can have these values:

Keyword	Description
Either Edge	Either rising or falling edge trigger, where the control signal changes either from 0 to a nonzero value, from a negative value to a positive value, or from a positive value to a negative value.
Rising Edge	Rising edge trigger, where the control signal changes from either from 0 to a positive value or from a negative value to a positive value.
Falling Edge	Falling edge trigger, where the control signal changes from either 0 to a negative value or from a positive value to a negative value.
Function Call	Function call triggered.

Each Stateflow block can only have one overall trigger type. See "Determining Overall Input Trigger Type" on page 3-18 for more information.

Apply the Changes

Click on the **Apply** button to save the properties. Click on the **Close** button to save the properties and close the dialog box.

Data Inputs from Simulink

Tasks to Define a Data Input from Simulink

These tasks describe how to add and define the necessary fields for a data input from Simulink:

- Add a data object choosing a chart as the parent of the data
- Choose **Input from Simulink** as the **Scope**
- Specify data attributes
- Apply and save the changes

Add a Data Object Choosing a Chart as the Parent

These steps describe how to add a data object:

1. Choose **Explore** from the graphics editor **Tools** menu to invoke the Explorer.
2. Select a chart object in the hierarchy that you want to be the data object's parent.

   You must explicitly choose a parent to create a data object. Choosing the Chart to be the parent determines that the data resides within the chart.
3. Choose **Data** from the Explorer **Add** menu. The **Data properties** dialog box is displayed.
4. Enter a name in the **Name** field.

Choose Input from Simulink as the Scope

Once you have chosen the chart as the parent, the choice of valid scopes includes: **Local**, **Input from Simulink**, **Output to Simulink, Temporary, or Workspace**.

Choose **Input from Simulink** as the **Scope** to enable access rights to Simulink and any offspring of the chart.

When you add a data input, each data input is represented on the Stateflow block by a Simulink input port. Multiple data inputs to the Stateflow block must be scalar (they cannot be vectorized).

Specify Data Attributes

If you want to change the defaults, you can specify data **Units**, **Type**, **Initial**, **Minimum**, and **Maximum** values.

Apply and Save the Changes

Click on the **Apply** button to save the properties. Click on the **Close** button to save the properties and close the dialog box.

Event Outputs to Simulink

Tasks to Define an Event Output to Simulink

These tasks describe how to add and define the necessary fields for an event output to Simulink:

- Add an event choosing a Chart as the parent of the data
- Choose **Output to Simulink** as the Scope
- Specify the **Trigger** type
- Apply and save the changes

Add an Event Choosing a Chart as the Parent

These steps describe how to add an event:

1. Choose **Explore** from the graphics editor **Tools** menu to invoke the Explorer.
2. Select a chart object in the hierarchy that you want to be the event's parent.

   You must explicitly choose a parent to create an event. Choosing the Chart to be the parent of the event enables receive rights to Simulink and to the chart and all its offspring.
3. Choose **Event** from the Explorer **Add** menu. The **Event properties** dialog box is displayed.
4. Enter a name in the **Name** field.

Choose Output to Simulink as the Scope

Once you have chosen the chart as the parent, the choice of valid scopes includes: **Local**, **Input from Simulink**, or **Output to Simulink**.

Choose **Output to Simulink** as the **Scope** to enable send rights to Simulink and any offspring of the chart and to enable receive rights to the Simulink model.

When you define an event to be an **Output to Simulink**, a Simulink output port is added to the Stateflow block. Output events from the Stateflow block to the Simulink model are scalar.

Apply the Changes

Click on the **Apply** button to save the properties. Click on the **Close** button to save the properties and close the dialog box.

Data Outputs to Simulink

Tasks to Define a Data Output to Simulink

These tasks describe how to add and define the necessary fields for a data output to Simulink:

- Add a data object choosing a chart as the parent of the data
- Choose **Output to Simulink** as the **Scope**
- Specify data attributes
- Apply and save the changes

Add a Data Object Choosing a Chart as the Parent

These steps describe how to add a data object:

1. Choose **Explore** from the graphics editor **Tools** menu to invoke the Explorer.
2. Select a chart object in the hierarchy that you want to be the data object's parent.

   You must explicitly choose a parent to create a data object. Choosing the Chart to be the parent determines that the data resides within the chart.
3. Choose **Data** from the Explorer **Add** menu. The **Data properties** dialog box is displayed.
4. Enter a name in the **Name** field.

Choose Output to Simulink as the Scope

Once you have chosen the chart as the parent, the choice of valid scopes includes: **Local**, **Input from Simulink**, or **Output to Simulink**.

Choose **Output to Simulink** as the **Scope** to enable access rights to Simulink and any offspring of the chart.

When you define a data object to be an **Output to Simulink**, a Simulink output port is added to the Stateflow block. Output data objects from the Stateflow block to the Simulink model are scalar.

Specify Data Attributes

If you want to change the defaults, you can specify data **Units**, **Type**, **Initial**, **Minimum**, and **Maximum** values.

Apply the Changes

Click on the **Apply** button to save the properties. Click on the **Close** button to save the properties and close the dialog box.

MATLAB Workspace

What Is the MATLAB Workspace?

The MATLAB workspace is the area of memory accessible from the MATLAB command line. The workspace maintains the set of variables built up during a MATLAB session.

See *Getting Started with MATLAB* and *Using MATLAB* for more information.

Using the MATLAB Workspace

You can define data objects to have a **Workspace Scope**. Any data objects defined with **Workspace Scope** must have value assigned in the MATLAB command window prior to starting a simulation.

Two commands, `who` and `whos`, show the current contents of the workspace. The `who` command gives a short list, while `whos` also gives size and storage information.

To delete all the existing variables from the workspace, enter `clear` at the MATLAB command line.

Using the **Workspace** data **Scope** is only meaningful for `sfun` simulation and RTW targets.

Defining the Interface to External Sources

What Are External Sources?

Any code that is not part of a Stateflow diagram, the Stateflow machine, nor the Simulink model is considered external. You can include external source code in either:

- the **Make Command** of the **Target Configuration** dialog box. (See "Make Command Customization" on page 9-23 for more information.)
- the **Custom Code** section of the **Target Configuration** dialog box. (See "Add Customized Code" on page 9-25)

See Chapter 3, "Events and Data" for conceptual information on events and data. See "Stateflow Explorer User Interface" on page 5-5 for more information on adding and defining events.

Exported Events

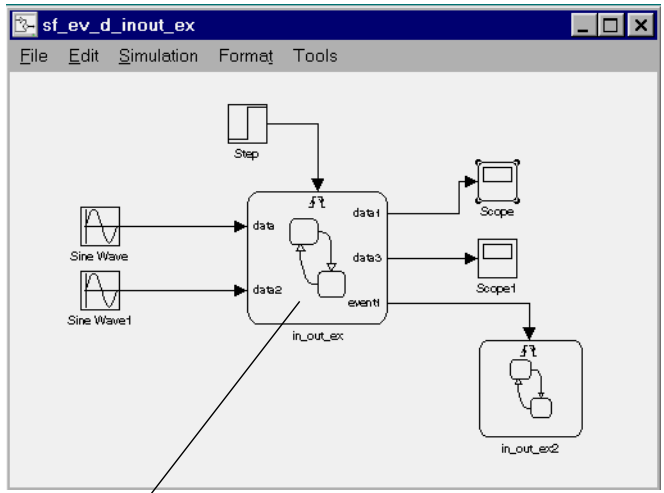
Consider a real world example to clarify when to define an **Exported** event. You have purchased a communications pager. There are a few people you want to be able to page you, so you give those people your personal pager number. These people now know your pager number and can call that number and page you whatever you might be doing. You do not usually page yourself, but you can do so. Telling someone the pager number does not mean they have heard and recorded the number. It is the other person's responsibility to retain the number.

Similarly, you may want an external source (outside the Stateflow diagram, the machine, and the Simulink model) to be able to broadcast an event. By defining an event's scope to be **Exported**, that event is made available to external sources for broadcast purposes. **Exported** events must be parented by the machine because the machine is the (highest) level in the Stateflow hierarchy that can interface to external sources. The machine also retains the ability to broadcast the **Exported** event. Exporting the event does not imply anything about what the external source does with the information. It is the responsibility of the external source to include the **Exported** event (in the manner appropriate to the source) to make use of the right to broadcast the event.

If the external source is another machine, then one machine defines an **Exported** event and the other machine defines the same event to be **Imported**.

Stateflow generates the appropriate export and import event code for both machines.

This example shows the format required in the external code source (custom code) to take advantage of an **Exported** event.



e is added and defined as an **Exported** event

e is imported in the external code source

external code source

```
void func_example(void)
{
extern void external_broadcast_e (void);
...
external_broadcast_e();
...
}
```

Stateflow generates this code:

```
void external_broadcast_e (void)
{
/* code based on the event
   definition
*/
...
{
```

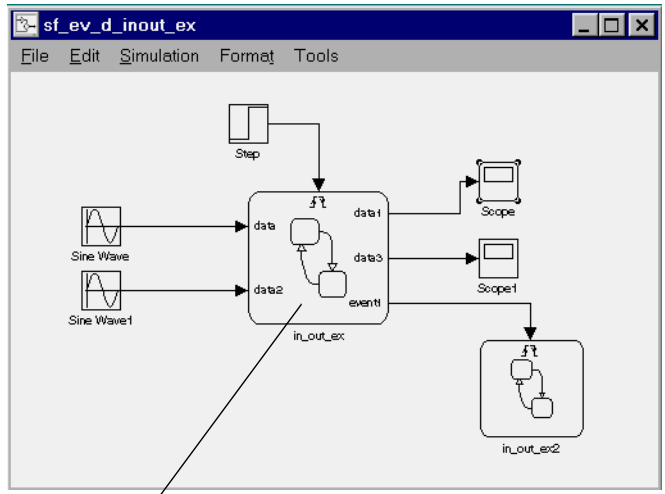
Imported Events

Consider the same pager example discussed for **Exported** events to clarify the use of **Imported** events. Someone buys a pager and indicates you may want to use this number to page them in the future. They tell you the pager number and you take note of the number by writing it down. You can then use the number to page that person. You cannot receive the page because you do not own the pager.

Similarly, you may want to broadcast an event that is externally (outside the Stateflow diagram, the machine, and the Simulink model) defined. By defining an event's scope to be **Imported**, the event can be broadcast anywhere within the hierarchy of that machine (including any offspring of the machine). An **Imported** event's parent is external. However, the event needs an 'adoptive' parent to resolve symbols for code generation. An **Imported** event's adoptive parent must be the machine because the machine is the (highest) level in the Stateflow hierarchy that can interface to external sources. Importing the event does not imply anything about what the external source does with the information. It is the responsibility of the external source to make the **Imported** event available (in the manner appropriate to the source) to make use of the right to broadcast the event.

If the external source is another machine then one machine defines an **Imported** event and the other machine defines the same event to be **Exported**. Stateflow generates the appropriate import and export event code for both machines.

This example shows the format required in the external code source (custom code) to make the event available.



e is exported in the external code source

external code source

```
void external_broadcast_e (void)

{
...
 }
```

e is added and defined as an **Imported** event

Stateflow generates this code for the **Imported** event:

```
extern void external_broadcast_e (void);
```

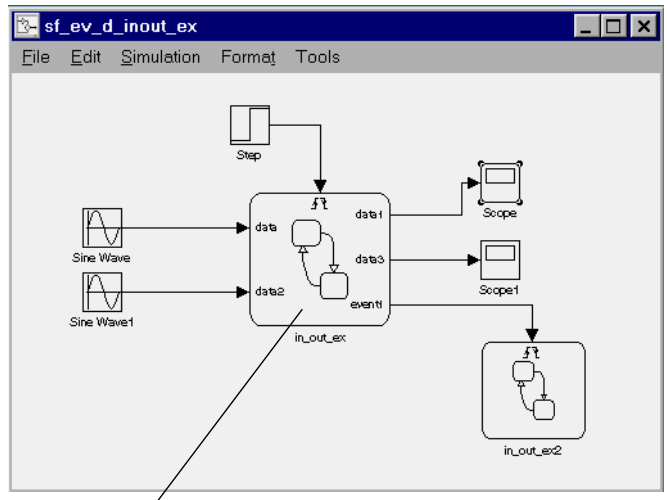
Exported Data

You may want an external source (outside the Stateflow diagram, the machine, and the Simulink model) to be able to access a data object. By defining a data object's scope to be **Exported**, that data is made accessible to external sources. **Exported** data must be parented by the machine because the machine is the (highest) level in the Stateflow hierarchy that can interface to external sources. The machine also retains the ability to access the **Exported** data object. Exporting the data object does not imply anything about what the external source does with the data. It is the responsibility of the external source to

include the **Exported** data object (in the manner appropriate to the source) to make use of the right to access the data.

If the external source is another machine, then one machine defines an **Exported** data object and the other machine defines the same data object to be **Imported**. Stateflow generates the appropriate export and import data code for both machines.

This example shows the format required in the external code source (custom code) to import an **Exported** data object.



`ext_data` is defined as imported in the external code source

`ext_data` added and defined as an **Exported** data

Stateflow generates this code:

```
int ext_data;
```

external code source

```
extern int ext_data;
void func_example(void)
{
...
ext_data = 123;
...
}
```

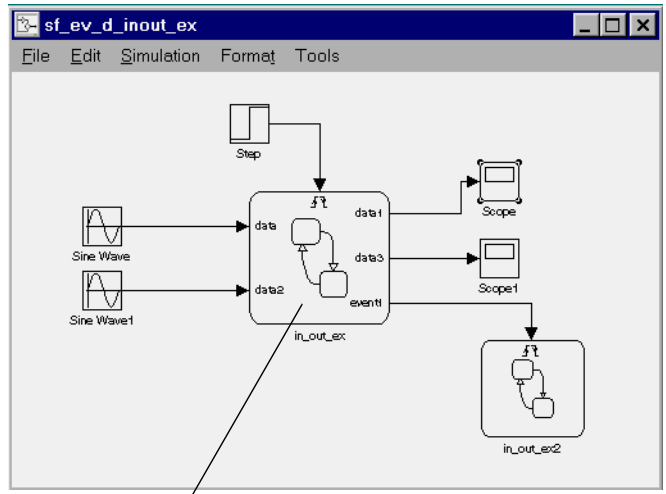
Imported Data

Similarly, you may want to access a data object that is externally (outside the Stateflow diagram, the machine, and the Simulink model) defined. By defining

a data's scope to be **Imported**, the data can be accessed anywhere within the hierarchy of that machine (including any offspring of the machine). An **Imported** data object's parent is external. However, the data object needs an 'adoptive' parent to resolve symbols for code generation. An **Imported** data object's adoptive parent must be the machine because the machine is the (highest) level in the Stateflow hierarchy that can interface to external sources. Importing the data object does not imply anything about what the external source does with the data object. It is the responsibility of the external source to make the **Imported** data object available (in the manner appropriate to the source) to make use of the right to access the data object.

If the external source is another machine, then one machine defines an **Imported** data object and the other machine defines the same data object to be **Exported**. Stateflow generates the appropriate import and export data code for both machines.

This example shows the format required if the data is **Imported** from an external code source (custom code).



`ext_data` added and defined as an **Imported** data

Stateflow generates this code:

```
extern int ext_data;
```

`ext_data` is defined as exported in the external code source

external code source

```
int ext_data;
void func_example(void)
{
...
}
```

5

Exploring and Searching

Overview 5-2

Stateflow Explorer User Interface 5-5

Example: Getting Started Adding Events and Data . . 5-16

Stateflow Finder User Interface 5-20

Searching for a Wildcard String 5-26

Searching for an Exact String Match 5-28

Searching for All Objects of a Specified Object Type . . 5-30

Searching Based on Multiple Conditions 5-32

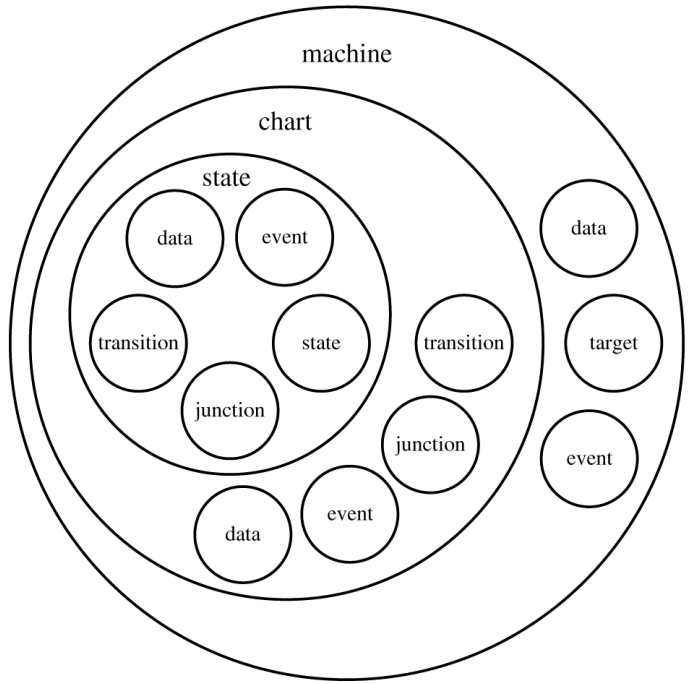
Searching for an Object to Edit Properties 5-34

Overview

Exploring and Searching

The Stateflow machine is the highest level in the Stateflow hierarchy. The object hierarchy beneath the Stateflow machine consists of combinations of the graphical and nongraphical objects. The data dictionary is the repository for all Stateflow objects. You can use the Stateflow Finder and Stateflow Explorer together to browse and make changes to data dictionary objects.

Stateflow data dictionary



The Explorer displays any defined events, data, and targets within an object hierarchy where machines, charts, and states are potential parents.

You can create, modify, and delete events, data, and target objects using the Explorer. You can also add events, data, and targets using the graphics editor **Add** menu. (See “Add Menu” on page 2-48 for more information.) If you add

data or events via the **Add** menu, the chart is automatically defined as the parent. If you add a target, the machine is defined as the parent. Targets can only be parented by the machine. If you want to change the parent of a data or event object, you must use the Explorer to do so. Similarly you must use the Explorer if you want to delete an event, data, or target object.

See "Customizing the Target" on page 9-20 for information on customizing the simulation target. See "Adding and Specifying a Stand-Alone Target" on page 9-16 for information on creating targets to generate code using the Stateflow Coder product.

The Finder displays a list of objects based on search criterion you specify. You can directly access the properties dialog box of any object in the search output display by double-clicking on that object.

The Finder, the Explorer, and the **Add** menu on the graphics editor provide access to the event, data, and target properties dialog boxes.

Where to Find More Information on Events and Data

See these sections for conceptual information on data and events: "Events" on page 3-5 and "Data" on page 3-22.

For information on data and events relevant to defining Stateflow interfaces, see Chapter 4, "Defining Stateflow Interfaces."

See "Adding and Specifying a Stand-Alone Target" on page 9-16 for information on defining standalone targets.

For information on customizing targets (including the simulation target) see "Customizing the Target" on page 9-20.

Typical Exploring Tasks

Exploring Stateflow diagrams can involve some or all of these tasks:

- Adding, defining, modifying, and deleting non-graphical objects (events, data, and targets).
- Exploring/browsing events, data, and target objects.
- Navigating through the hierarchy of open machines, charts, and states.

See "Example: Getting Started Adding Events and Data" on page 5-16 for an example of adding and defining **Local** events and data parented by the chart.

See these sections for information on adding events and data in more complex contexts:

- "Event Inputs from Simulink" on page 4-16
- "Data Inputs from Simulink" on page 4-18
- "Event Outputs to Simulink" on page 4-20
- "Data Outputs to Simulink" on page 4-22
- "Defining the Interface to External Sources" on page 4-25

Typical Searching Tasks

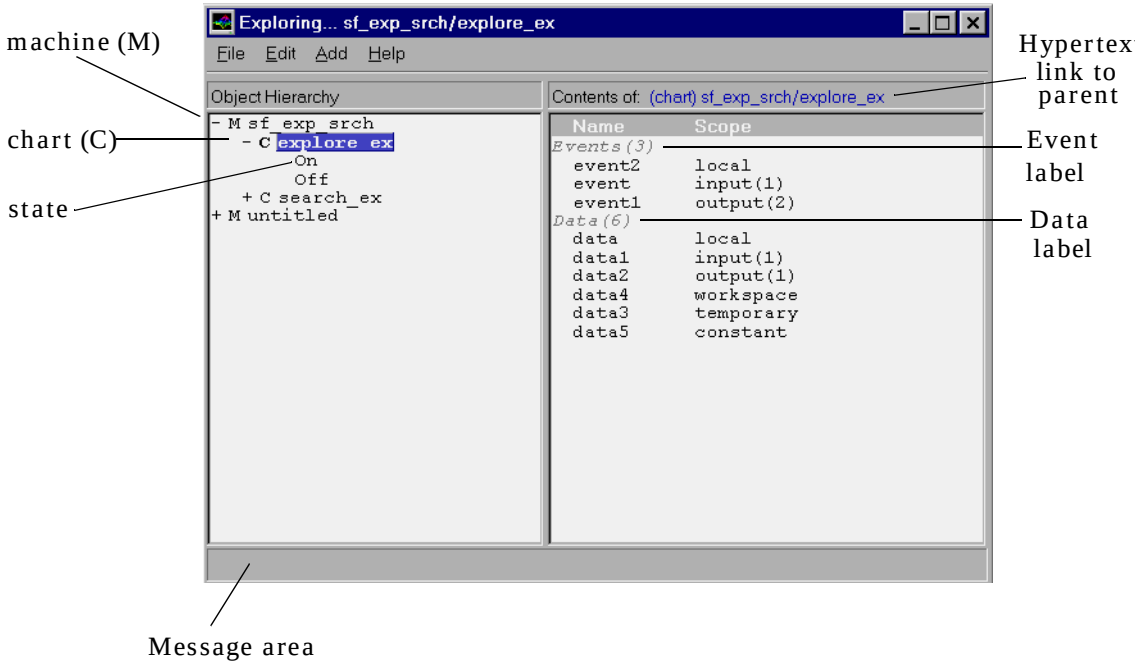
Using the Finder you can:

- Search the data dictionary for objects. Searches are performed across all of the Stateflow diagrams within a Simulink model (Stateflow machine).
  - Search for all objects that contain a specified string (or subset of a string) in the label, name, description, or any combination thereof, of the object (wildcards/regular expressions are supported). See "Searching for an Exact String Match" on page 5-28.
  - Search for all objects that have an exact word match in the label, name, description, or any combination thereof, of the object. See "Searching for an Exact String Match" on page 5-28.
  - Search for all objects of a specified object type. See "Searching for All Objects of a Specified Object Type" on page 5-30.
- Perform certain actions on objects listed in the search results display area.
  - Edit object properties. See "Searching for an Object to Edit Properties" on page 5-34
  - Automatically select objects in their Stateflow diagrams.

Stateflow Explorer User Interface

Explorer Main Window

This is the Explorer main window showing the object hierarchy of an example chart (explore_ex):



Object Hierarchy

The **Object Hierarchy** (machines (M), charts (C), and states) is displayed in the left-hand pane. A '+' character indicates that the hierarchy (machine (M), chart (C), or state) can be expanded by double-clicking on that entry (or by clicking on the '+' character. A '-' character indicates there is nothing to expand. As you move the cursor over an **Object Hierarchy** entry, it is shown in blue and indicates the cursor focus. The focus tells you which object would be selected if you clicked the mouse at that moment. Clicking on an entry in the **Object Hierarchy** selects that entry. Selection is indicated by a blue background and bold text.

Contents of:

Events, data, and target objects parented by the currently selected object in the **Object Hierarchy** are displayed in the **Contents of:** pane. Each type of object has a heading label that also indicates the number of objects of that type defined.

These are the possible parent and object combinations:

	Machine (M)	Chart (C)	State
Event	yes	yes	yes
Data	yes	yes	yes
Target	yes	no	no

Targets are parented exclusively by machines. (Although all other combinations are valid, there are guidelines describing how **Scope** affects choice of parent and vice-versa.) The default `sfun` simulation target is automatically defined for every machine. If you have a Real-Time Workshop license, an RTW target is also automatically added:

- When you select **Open RTW Target** from the graphics editor **Tools** menu
- If you build a target that includes a Stateflow machine using Real-Time Workshop

For convenience, a hypertext link to the parent of the currently selected object in the **Object Hierarchy** is included following the **Contents of:** label. Click on the hypertext link to bring that object to the forefront.

File Menu

Select **Close (Ctrl+W)** from the **File** menu to close the Explorer window.

Edit Menu

Select **Cut (Ctrl+X)** from the **Edit** menu to delete the selected data, event, or target object in the **Contents of:** pane.

Add Menu

The **Add** menu provides these options:

Menu Choice	Description
Event (Ctrl+E)	Creates a new event at the selected level in the object hierarchy; displays an **Event properties** dialog box.
Data (Ctrl+D)	Creates a new data object at the selected level in the object hierarchy; displays a **Data properties** dialog box.
Target (Ctrl+T)	Creates a new untitled standalone target at the selected machine level in the object hierarchy (a target's parent must be a machine; this menu item is disabled if a chart or state parent type is selected); displays an untitled **Target Configuration properties** dialog box.

Help Menu

The Help menu provides these options:

Menu Choice	Description
Topics (HTML)	Displays Stateflow online help in an HTML browser. An online version of the *Stateflow User's Guide* is one of the options available from the online help.
Help Desk (HTML)	Displays the MATLAB Help Desk in an HTML browser.
About Stateflow	Displays a pop-up window with software version information.

Moving Objects/Changing Parent

To create desired behavior you may need to change the parent of an event, data, or target object.

Objects in the **Contents of:** pane can be moved in the hierarchy to change an object's parent. Click and drag an object from the **Contents of:** pane to a new location in the **Object Hierarchy** pane to change its parent. If the object is the current parent, an X with a circle around it is displayed (indicating this is an invalid operation). If you move an object to a level in the hierarchy that does not support that object's current **Scope** property, the **Scope** is changed to **Local**.

Moving Objects/Changing Index and Port Order

To ensure proper ordering of event and/or data **Input from** or **Output to Simulink** you may need to move some of these objects in the Explorer.

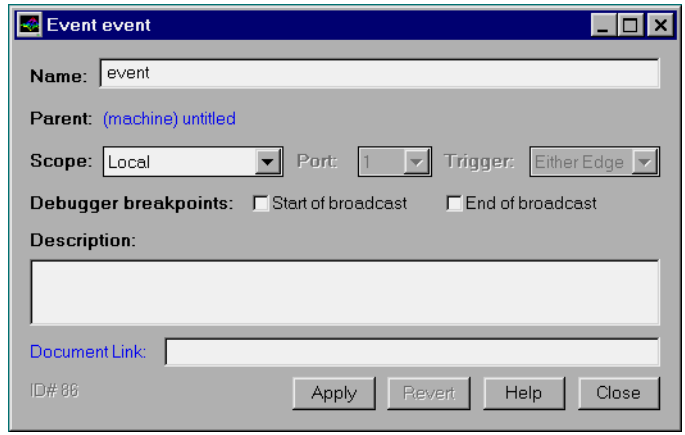
Click and drag a data object with **Input from** or **Output to Simulink Scope** to a new position in the **Contents of:** pane **Data list** to change its port number. Click and drag an event **Input from** or **Output to Simulink Scope** to a new position in the **Contents of:** pane **Event list** to change its index number.

Deleting Objects

Select the object in the **Contents of:** pane and press the **Delete** key or select **Cut (Ctrl+X)** from the **Edit** menu to delete an object.

Event Properties Dialog Box

This is a new **Event properties** dialog box.



Name

Enter the name of the event object in this field.

Parent

The event's parent is listed as a hyptertext link in a read-only field. The object type is in parenthesis followed by the name of the object.

Clicking on the parent (in blue) displays and changes focus to either the Simulink model (parent is the machine) or the graphics editor (parent is either the chart or a state), depending on the parent.

See these sections for more information: "Event Parent Types" on page 3-7, "Events, Hierarchy, and Scope" on page 3-9.

Scope

An event's **Scope** defines to what extent the object is known relative to the Stateflow block, the Simulink model, and sources external to the Simulink model.

By default the **Scope** is **Local**. The **Scope** options are displayed in a pull-down menu. The parent (machine, chart, or state) dictates the choice of **Scope**. If the **Scope** you desire is not an available option for that parent type, you may need to move the event so that it is parented by an object that supports that **Scope**.

If you move an object to a level in the hierarchy that does not support that object's current **Scope** property, the **Scope** is changed to **Local**.

These are all possible options for the **Scope** property:

- **Local**
- **Input from Simulink**
- **Output to Simulink**
- **Exported**
- **Imported**

See "Event Scope" on page 3-10 for more information.

Port

The **Port** field is relevent for events with **Output to Simulink Scope**. The **Port** number options are displayed in a pull-down menu.

See "Event Ports" on page 3-16 for more information.

Index

The **Index** field is relevant for events with **Input from Simulink Scope**. The **Index** number options are displayed in a pull-down menu.

See "Event Indices" on page 3-16 for more information.

Trigger

The **Trigger** field is relevant for events with **Input from Simulink** or **Output to Simulink Scope** (otherwise the field is grayed out). The **Trigger** options are displayed in a pull-down menu.

A trigger defines how a Stateflow block's input and output events are handled in the context of their Simulink model. Each Stateflow block can only have one overall **Trigger** type. The **Trigger** type indicates what kind of signal has meaning for the input or output event. **Trigger** types fall into two general categories: edge or function call triggered.

The **Trigger** property can have these values:

- **Either Edge**
- **Rising Edge**
- **Falling Edge**
- **Function Call**

For more information, see these sections: “Event Triggers” on page 3-17 and “Determining Overall Input Trigger Type” on page 3-18.

Debugger Breakpoints

Click in the checkbox (display a checkmark) to set a breakpoint either at the start of the event broadcast (**Start of Broadcast**) or at the end of the event broadcast (**End of Broadcast**).

Description

Click in the text field area labeled **Description** and enter any textual comments or descriptions for the object.

Document Link

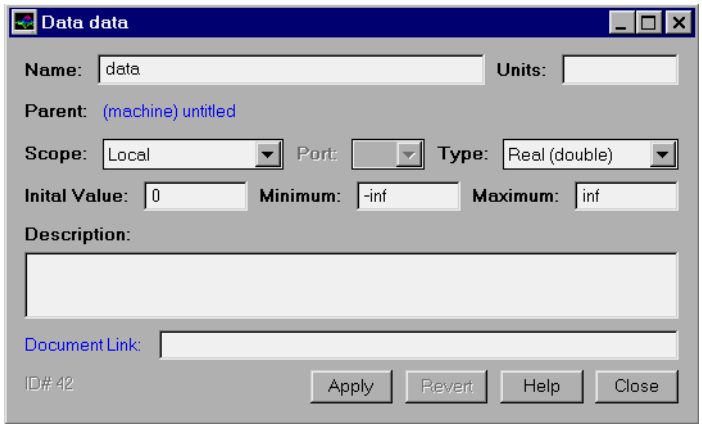
Enter a web URL address or a general MATLAB command in the **Document Link** field. Examples of links are: `www.mathworks.com`, `mailto:email_address`, `edit/spec/data/speed.txt`.

Apply, Revert, and Close Buttons

Click on the **Apply** button to save the current values. Click on the **Revert** button to go back to the original values after changes have been applied but before the dialog box has been closed. The **Revert** button is grayed out until a change has been applied. Click on the **Close** button to save the current values and close the dialog box.

Data Properties Dialog Box

This is an untitled data object properties dialog box.



Name

Enter the name of the data object in this field.

Units

Units is a user-entered string to describe the units of the data type.

Parent

The data object's parent is listed in a read-only field. The parent is the object in the **Object Hierarchy** display that was selected when the data object was created. The object type is in parenthesis followed by the name of the object.

Clicking on the parent (in blue) displays and changes focus to either the Simulink model (parent is the machine) or the graphics editor (parent is either the chart or a state), depending on the parent.

See these sections for more information: "Data Parent Types" on page 3-25 and "Data, Hierarchy, and Scope" on page 3-26.

Scope

A data object's scope defines to what extent the object is known relative to the Stateflow block, the Simulink model, and sources external to the Simulink model.

By default the **Scope** is **Local**. The **Scope** options are displayed in a pull-down menu. The options displayed depend on the parent.

If the **Scope** you desire is not an available option for that parent type, you may need to move the event so that it is parented by an object that supports that **Scope**.

These are the options for the **Scope** property:

- **Local**
- **Input from Simulink**
- **Output to Simulink**
- **Temporary**
- **Workspace**
- **Constant**
- **Exported**
- **Imported**

See “Data Scope” on page 3-27 for more information.

Port

The **Port** field is relevant for data objects with **Input from Simulink** or **Output to Simulink Scope** (otherwise the field is grayed out). The **Port** number options are displayed in a pull-down menu.

See “Data Ports” on page 3-32 for more information.

Type

Type is the data type of the data object. The default data type is **Real (double)**. These **Type** options are displayed in a pull-down menu:

- **Real (double)**
- **Real (float)**
- **Integer (long)**
- **Integer (short)**
- **Boolean (unsigned char)**
- **Word (unsigned long)**
- **Word (unsigned short)**
- **Byte (unsigned char)**

Initial Value

The **Initial Value** is the user-entered intial value of the data object (0 by default).

Minimum

The **Minimum** is the user-entered smallest scalar value of the data object (-inf, negative infinity, by default).

Maximum

The **Maximum** is the user-entered largest scalar value of the data object (inf, negative infinity, by default).

Description

Click in the text field area labeled **Description** and enter any textual comments or descriptions for the object.

Document Link

Enter a web URL address or a general MATLAB command in the **Document Link** field. Examples of links are: `www.mathworks.com`, `mailto:email_address`, `edit/spec/data/speed.txt`.

Apply, Revert, and Close Buttons

Click on the **Apply** button to save the current values. Click on the **Revert** button to go back to the original values after changes have been applied but before the dialog box has been closed. The **Revert** button is grayed out until a change has been applied. Click on the **Close** button to save the current values and close the dialog box.

Example: Getting Started Adding Events and Data

Task Overview

The flexibility and power provided by parenting and scoping events and data can be overwhelming and complex to begin with. The simplest case to consider is:

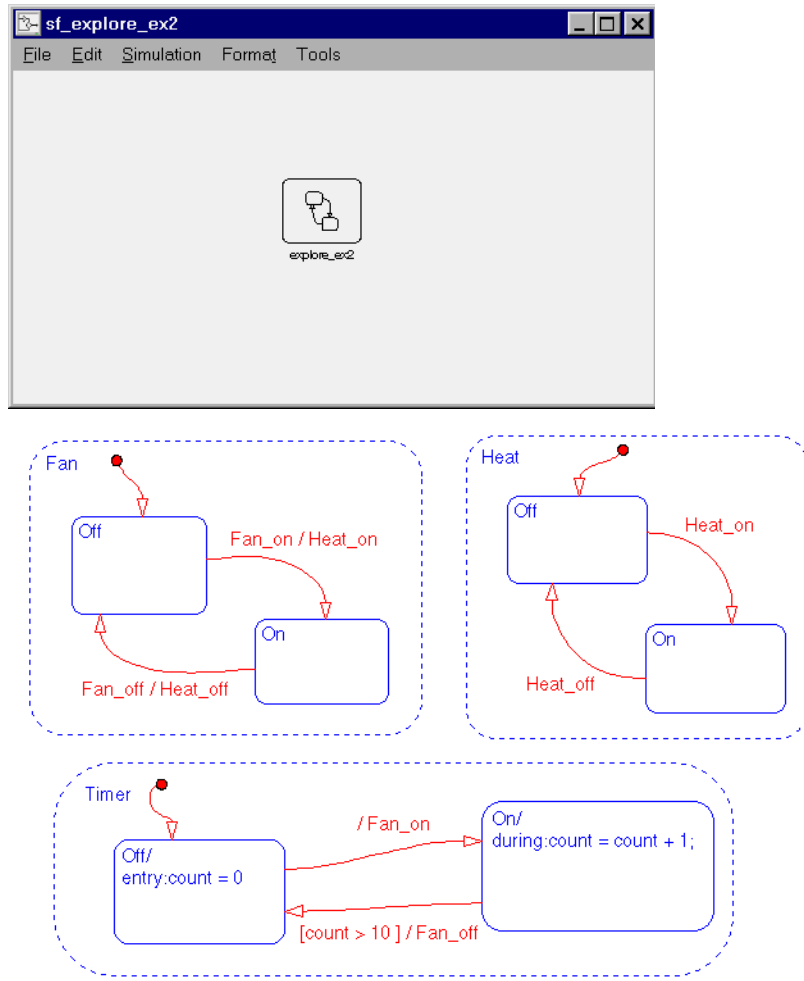
- Create a Simulink model containing a single Stateflow diagram
- Use the Simulink implicit event broadcast to awaken the Stateflow diagram
- Define events and data parented by the chart
- Define events and data scope as **Local**
- Broadcast any **Local** events within the Stateflow diagram

Events and data are typically added using the Explorer. The graphics editor **Add** menu provides a shortcut method to quickly add new events and data. Using the **Add** menu creates the object with the chart as the parent. By default the object's scope is **Local**.

Solution Model and Stateflow Diagram

The Simulink model contains a single Stateflow diagram and uses the Simulink implicit event broadcast to awaken the Stateflow diagram and trigger transitions labeled to be valid on any event broadcast. Use the default chart **Update method** that is **Triggered or Inherited** (set in the **Chart properties** dialog box) to achieve this behavior.

This Simulink model and Stateflow diagram are used in this section.



What Are the Events?

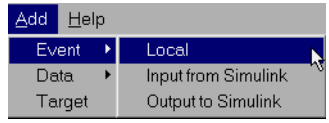
There are four **Local** events (`Fan_on`, `Fan_off`, `Heat_on`, and `Heat_off`) in the Stateflow diagram. The `Timer` parallel (AND) superstate broadcasts the **Local** event `Fan_on`. The transition from `Fan.Off` to `Fan.On` that is triggered by the broadcast of **Local** event `Fan_on` broadcasts the transition action `Heat_on`. The

transition from `Heat.Off` to `Heat.On` is triggered by the broadcast of the **Local** event `Heat_on`.

Add Local Events

When you create the graphical objects in the Stateflow diagram, you use the event names as part of the transition labels. Adding them as labels does not define them as events in the data dictionary. Use either the graphics editor **Add** menu or the Explorer to add and define events.

1. Select **Event** from the graphics editor **Add** menu. Select **Local** from the **Event** submenu.



2. Replace the contents of the name field with the name `Fan_on`.

3. Click on the **Close** button to save the changes and close the **Event properties** dialog box.

4. Select **Explore** from the graphics editor **Tools** menu to display the Explorer.

5. Select the chart in the **Object Hierarchy** that you want to be the event's parent.

6. Choose **New** and then **Event** from the Explorer **File** menu. The **Event properties** dialog box is displayed.

7. Replace the contents of the name field with the name `Fan_off`.

8. Click on the **Close** button to save the changes and close the **Event properties** dialog box.

9. Add, define, and save the remaining **Local** events: `Heat_on`, and `Heat_off` using either the **Add** menu or the Explorer.

What Is the Data?

There is one **Local** data object (`count`) in the Stateflow diagram. `count` is used as a timer variable. To start with, `count` is zero and `Timer.Off` is active. When

any event occurs, the transition from `Timer.Off` to `Timer.On` is valid and the **Local** event `Fan_on` is broadcast. When count becomes greater than ten, the transition from `Timer.On` to `Timer.Off` is valid and the **Local** event `Fan_off` is broadcast.

Adding Local Data

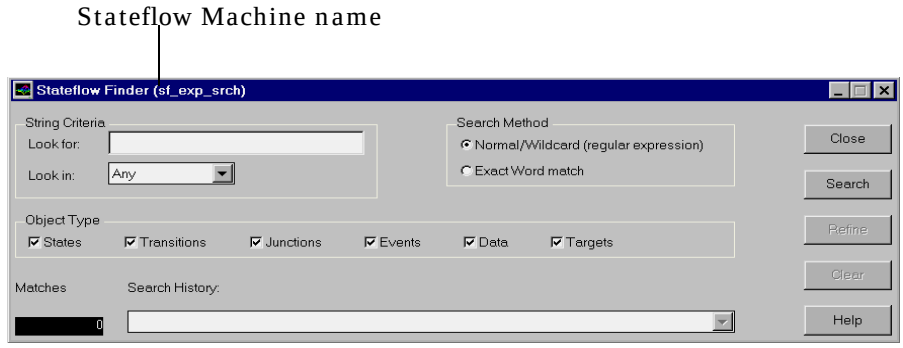
When you create the graphical objects in the Stateflow diagram, you use the data name as part of state actions and a transition label. Adding them as labels and actions does not define them as data objects in the data dictionary. Use either the graphics editor **Add** menu or the Explorer to add and define data.

1. Select **Explore** from the graphics editor **Tools** menu to display the Explorer.
2. Select the chart in the **Object Hierarchy** that you want to be the data object's parent.
3. Choose **New** and then **Data** from the Explorer **File** menu. The **Data properties** dialog box is displayed.
4. Replace the contents of the name field with the name `count`.
5. Set the **Minimum:** field to 0 and the **Maximum:** field to 10.
6. Click on the **Close** button to save the changes and close the **Data properties** dialog box.

Stateflow Finder User Interface

Finder Dialog Box

The Finder operates on a machine. This is the Finder dialog box:



String Criteria

You specify the string by entering the text to search for in the **Look for:** text box. The search is case sensitive. All text fields are included in the search by default. Alternatively, you can search in specific text fields by using the drop down **Look in:** list box to choose one of these options:

- Any

  Search the state and transition labels, object names, and descriptions of the specified object types for the string specified in the **Look for:** field.

- Label

  Search the state and transition labels of the specified object types for the string specified in the **Look for:** field.

- Name

  Search the name fields of the specified object types for the string specified in the **Look for:** field.

- Description

  Search the description fields of the specified object types for the string specified in the **Look for:** field.

Search Method

By default the **Search Method** is **Normal/Wildcard** (regular expression). Alternatively, you can click on the **Exact Word match** option if you are searching for a particular sequence of one or more words.

A regular expression is a string composed of letters, numbers, and special symbols that defines one or more strings. Some characters have special meaning when used in a regular expression while other characters are interpreted as themselves. Any other character appearing in a regular expression is ordinary, unless a \ precedes it.

These are the special characters supported by Stateflow:

Character	Description
^	Start of string
$	End of string
.	Any character
\	Quote the next character
*	Match zero or more
+	Match one or more
[ ]	Set of characters

Object Type

Specify the object type(s) to search by toggling the radio boxes. A check mark indicates that the object is included in the search criteria. By default, all object types are included in the search criteria. **Object Types** include:

- States
- Transitions
- Junctions
- Events
- Data
- Targets

Search Button

Click on the **Search** button to initiate the search operation. The data dictionary is queried and the results are listed in the display area.

Matches

The **Matches** field displays the number of objects that match the specified search criteria.

Refine Button

After the results of a search are displayed, enter additional search criteria and click on the **Refine** button to narrow the previously entered search criteria. An ampersand(&) is prepended to the search criteria in the **Search History:** field to indicate a logical AND with any previously specified search criteria.

Search History

The **Search History** text box displays the current search criteria. Click on the pull-down list to display search refinements. An ampersand is prepended to the search criteria to indicate a logical AND with any previously specified search criteria. You can undo a previously specified search refinement by selecting a previous entry in the search history. By changing the **Search History** selection you force the Finder to use the specified criteria, as the current, most refined, search output.

Clear Button

Click the **Clear** button to clear any previously specified search criteria. Results are removed and the search criteria is reset to the default settings.

Close Button

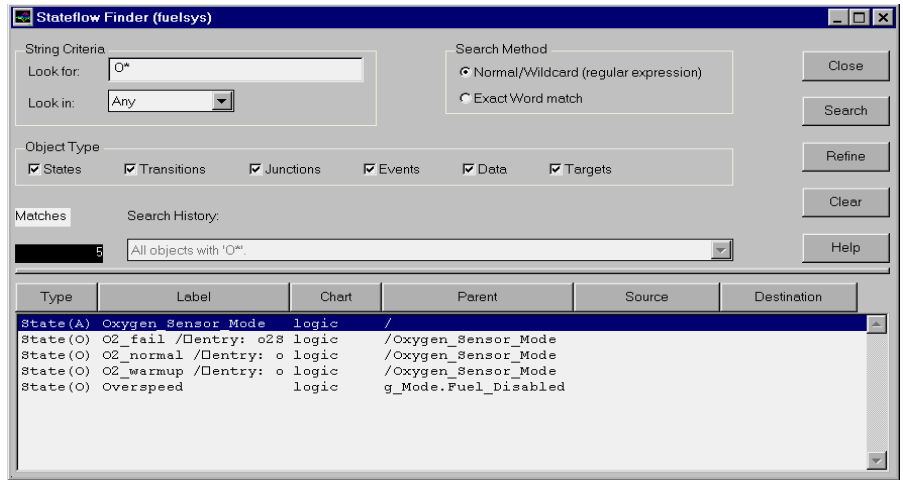
Click the **Close** button to close the Finder.

Help Button

Click the **Help** button to display the Stateflow online help in an HTML browser window.

Finder Display Area

The Finder display area looks like this:



The display area is divided into these fields:

Field	Description
Type	The object type is listed in this field. States with exclusive (OR) decomposition are followed by an (O). States with parallel (AND) decomposition are followed by (A).
Label	The string label of the object is listed in this field.
Chart	The title of the Stateflow diagram (Stateflow block) is listed in this field.
Parent	This object's parent in the hierarchy.
Source	Source object of a transition.
Destination	Destination object of a transition.

All fields are truncated to maintain column widths. The **Parent**, **Source**, and **Destination** fields are truncated from the left so that the name at the end of

the hierarchy is readable. The entire field contents, including the truncated portion, is used for resorting.

Each field label is also a button. Click on the button to have the list sorted based on that field. If the same button is pressed twice in a row, the sort ordering is reversed.

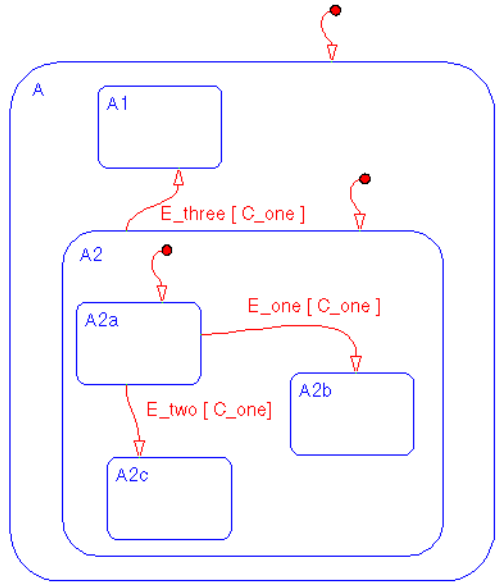
The Finder can be resized vertically to display more output rows, but cannot be expanded horizontally.

Click on a graphical entry to highlight that object in the graphical editor window. Double-click on an entry to invoke the **Property** dialog box for that object.

Representing Hierarchy

The Finder displays **Parent**, **Source**, and **Destination** fields to represent the hierarchy. The Stateflow diagram is the root of the hierarchy and is represented by the / character. Each level in the hierarchy is delimited by a . character. The **Source** and **Destination** fields use the combination of the ~ and the . characters to denote that the state listed is relative to the **Parent** hierarchy.

Using this Stateflow diagram as an example,



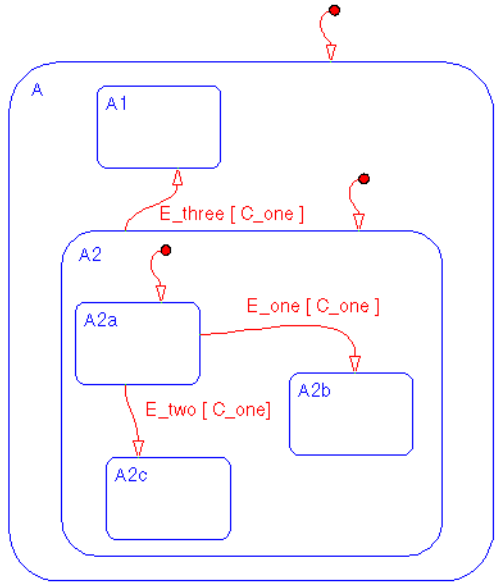
what are the values for the **Parent**, **Source,** and **Destination** fields for the transition from A2a to A2b?

The transition is within state A2. State A2's parent is state A and state A's parent is the Stateflow diagram itself. /A.A2 is the notation for state A2a's parent. State A2a is the transition source and state A2b is the destination. These states are at the same level in the hierarchy. ~.A2a is the relative hierarchy notation for the source of the transition. The full path is /A.A2.A2a. The relative hierarchy notation for the destination of the transition is ~.A2b. The full path is /A.A2.A2b.

Searching for a Wildcard String

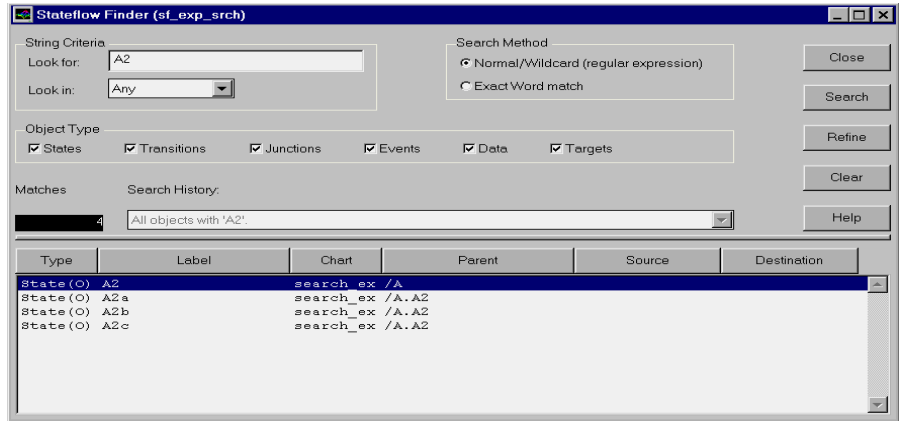
Example

This example Stateflow diagram is used to show how to find all object labels containing a specified wildcard string.



1. Select **Find...** from the **Tools** menu of the Stateflow graphics editor window to invoke the Finder.
2. Click in the **Look for:** text field and enter A2.
3. Click on and enable the **Normal/Wildcard** radio box.

4 Click on the **Search** button. These are the results of the search.

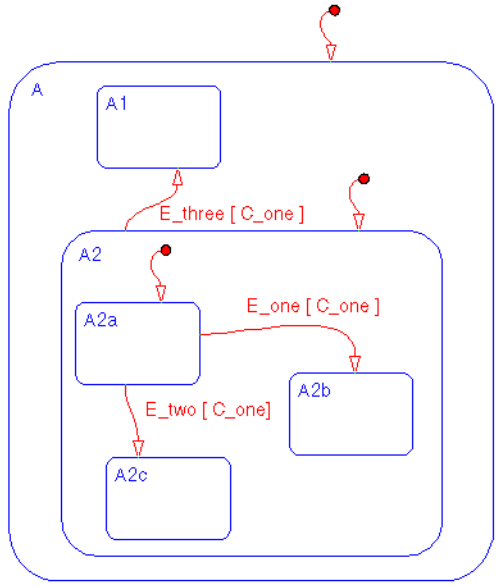


There are four matches. The **Search History** shows that the current search criterion is `All objects with 'A2'`. The display shows the matches are all exclusive OR (O) states.

Searching for an Exact String Match

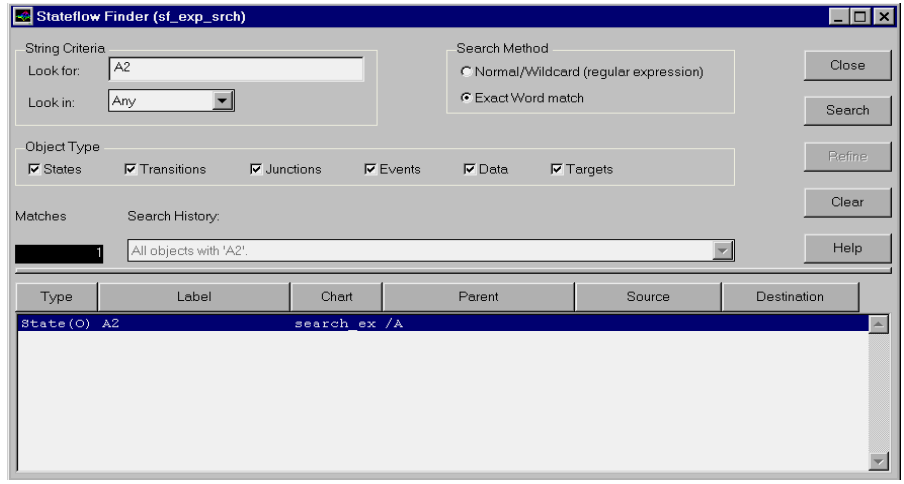
Example

This example Stateflow diagram is used to show how to find all objects with an exact string match in the label.



1. Select **Find...** from the **Tools** menu of the Stateflow graphics editor window to invoke the Finder.
2. Click in the **Look for** text field and enter A2.
3. Click on and enable the **Exact word match** radio button.

4 Click on the **Search** button. These are the results of the search.

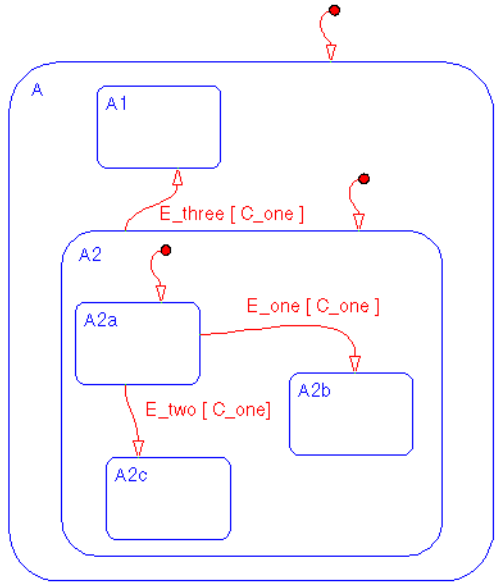


There is one match. The **Search History** shows that the current search criterion is `All objects with 'A2'`. The display shows the match is an exclusive (O) state labeled `A2` in a Stateflow diagram titled `search_ex`. The parentage of that state is `/A`.

Searching for All Objects of a Specified Object Type

Example

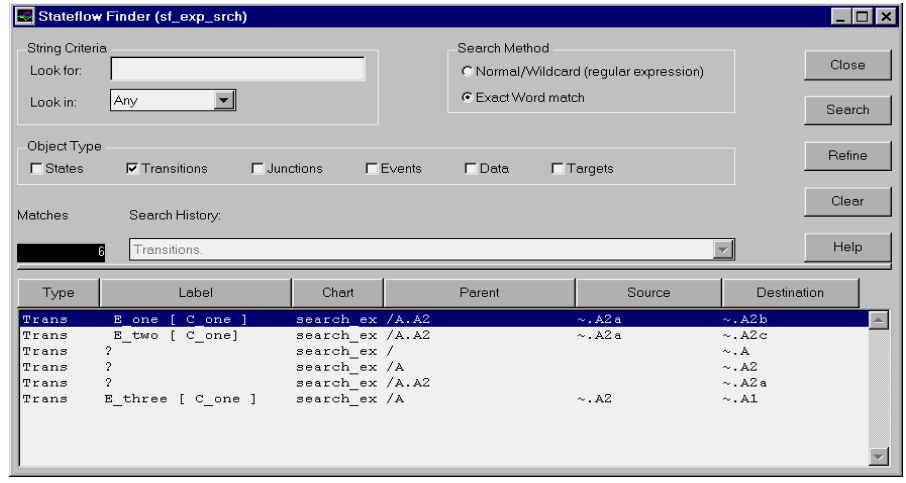
This Stateflow diagram is used to show how to search for all objects of a specified type.



1. Select **Find...** from the **Tools** menu of the Stateflow graphics editor window to invoke the Finder.

2. Ensure that the **Transitions** object type is checked. A checkmark indicates that object is included in the search criteria. By default, all object types are included in the search criteria. Deselect all other object types.

3 Click on the **Search** button. These are the results of the search.

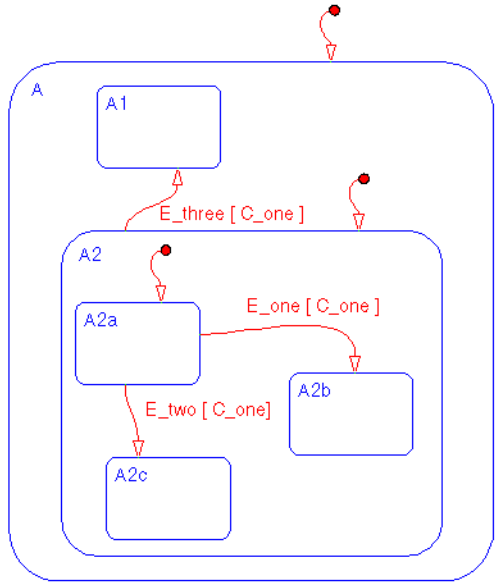


There are six matches. The **Search History** shows that the current search criterion is `Transitions`. The display shows the matches.

Searching Based on Multiple Conditions

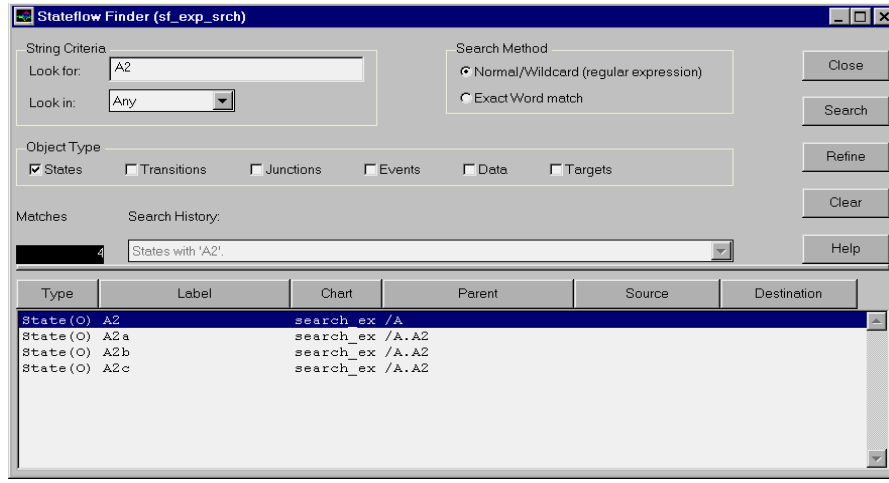
Example

This Stateflow diagram is used to show how to search based on multiple conditions.



1. Select **Find...** from the **Tools** menu of the Stateflow graphics editor window to invoke the Finder.
2. Click in the **Look for** text field and enter A2.
3. Click on and enable the **Normal/Wildcard** radio box.
4. Ensure that the **States** object type is checked. A checkmark indicates that object is included in the search criteria. By default, all object types are included in the search criteria. Deselect all other object types.

5 Click on the **Search** button. These are the results of the search.

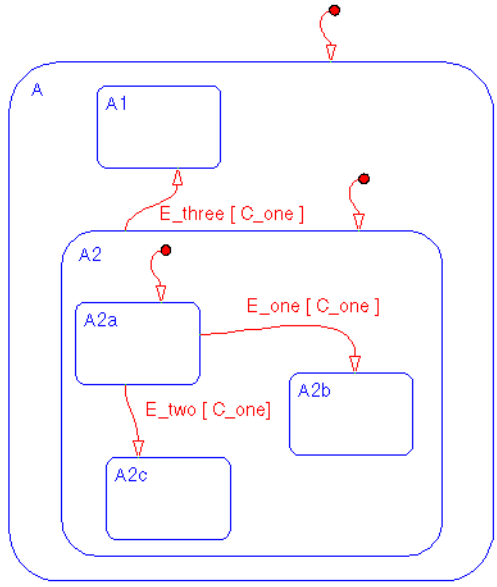


There are four matches. The **Search History** shows that the current search criterion is `States with 'A2'`. The display shows the matches.

Searching for an Object to Edit Properties

Example

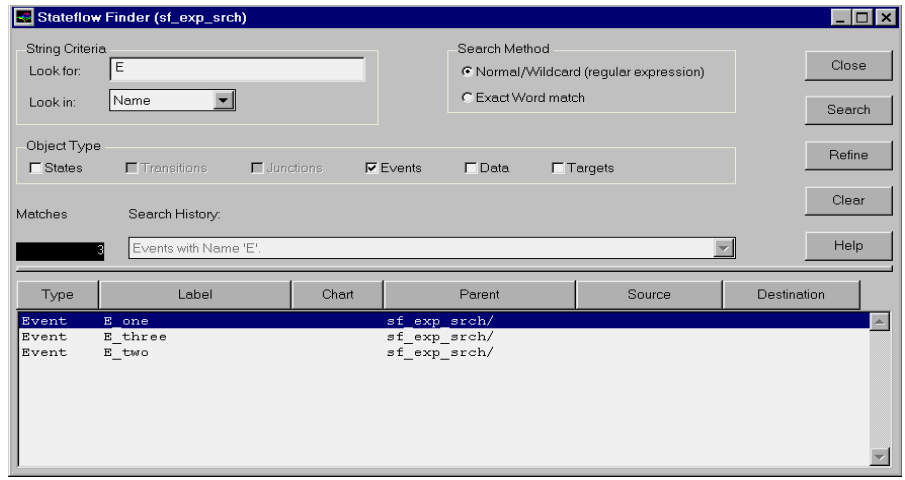
This Stateflow diagram is used to show how to search for an object and edit its properties.



1. Select **Find...** from the **Tools** menu of the Stateflow graphics editor window to invoke the Finder.
2. Click in the **Look for:** text field and enter *E*.
3. Choose *Name* from the **Look In** field pull-down menu.
4. Click on and enable the **Normal/Wildcard** radio box.
5. Ensure that the **Events** object type is checked. Event and data objects must already be defined using either the Explorer or the graphics editor **Add** menu. A checkmark indicates that object is included in the search criteria.

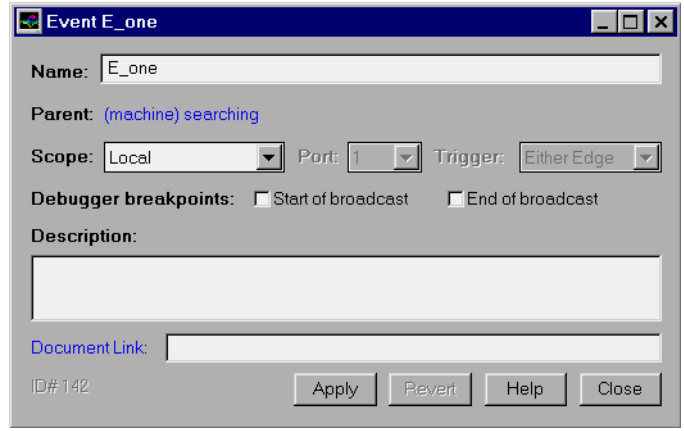
By default, all object types are included in the search criteria. Deselect all other objects.

6 Click on the **Search** button. These are the results of the search.



There are three matches. The **Search History** shows that the current search criterion is `Events with name 'E'`. The display shows the matches.

7 Double-click on event `E_one` to display the Event properties dialog box.



8 Add a description. Click on the Close button to save the changes and close the **Event properties** dialog box. An event's parent can be changed only through the Explorer.

6

Notations

Overview 6-2

States 6-10

Transitions 6-17

Connective Junctions 6-32

History 6-40

Action Language 6-42

Overview

What Is Meant by Notation?

A notation defines a set of objects and the rules that govern the relationships between those objects. Stateflow notation provides a common language to communicate the design information conveyed by a Stateflow diagram.

Stateflow notation consists of:

- A set of graphical objects
- A set of nongraphical text-based objects
- Defined relationships between those objects
- Keywords
- Action language

Motivation Behind the Notation

Chapter 2, “Using the Graphics Editor,” and Chapter 5, “Exploring and Searching,” discuss how to use the product to create the various objects. Knowing how to create the objects is the first step to designing and implementing a Stateflow diagram. The next step is understanding and using the notation to create a well-designed and efficient Stateflow diagram.

This chapter focuses on the notation: the supported relationships amongst the graphical objects and the action language that dictates the actions that can be associated with states and transitions. The Stateflow notation supports many different ways of representing desired system behavior. The representation you choose directly affects the efficiency of the generated code. You can use the notation to optimize for these code attributes:

- Speed
- Code size
- Memory use

See “Using the Notation Efficiently” on page 9-7 for more information.

How Is the Notation Checked?

The parser evaluates the graphical and nongraphical objects in each Stateflow machine against the supported Stateflow notation and the action language syntax. Errors are displayed in informational popup windows. See "Parsing" on page 9-11 for more information.

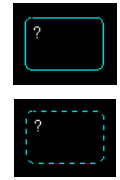
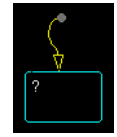
Some aspects of the notation are verified at runtime. Using the Debugger you can detect runtime errors such as:

- State inconsistencies
- Conflicting transitions
- Data range violations
- Cyclic behavior

You can modify the notation to resolve runtime errors. See Chapter 10, "Debugging," for more information on debugging runtime errors.

What Are the Graphical Objects?

These are the graphical objects in the notation that are on the toolbar:

Name	Notation	Toolbar Icon
State		
History junction		
Default transition		
Connective junction		

A transition is a curved line with an arrowhead that links one graphical object to another. Either end of a transition can be attached to a source and a destination object. The *source* is where the transition begins and the *destination* is where the transition ends.

What Are the Nongraphical Objects?

Event and data objects do not have graphical representations. These objects are defined using the Stateflow Explorer. See Chapter 3, “Events and Data,” and Chapter 5, “Exploring and Searching,” for more information.

What Is the Data Dictionary?

The data dictionary is a database containing all the information about the graphical and nongraphical objects. Data dictionary entries for graphical objects are created automatically as the objects are added and labeled. You explicitly define nongraphical objects in the data dictionary by using the Explorer. The parser evaluates entries and relationships between entries in the data dictionary to verify the notation is correct.

Keywords

These Stateflow keywords have special meaning in the notation:

Keyword	Shorthand	Meaning
`change(`*`data_name`*`)`	`chg(`*`data_name`*`)`	A condition function that is evaluated as true when the value of *data_name* specified as the argument changes.
`during:`	`du`	Actions that follow are executed as part of a state's `during` action.
`entry:`	`en`	Actions that follow are executed as part of a state's `entry` action.
`entry(`*`state_name`*`)`	`en(`*`state_name`*`)`	Generates a local event when the specified *state_name* is entered.
`exit`	`ex`	Actions that follow are executed as part of a state's `exit` action.

Keyword	Shorthand	Meaning
`exit(`*`state_name`*`)`	`ex(`*`state_name`*`)`	Generates a local event when the specified *state_name* is exited.
`in(`*`state_name`*`)`	none	A condition function that is evaluated as true when the *state_name* specified as the argument is active.
`on` *event_name*	none	Actions that follow are executed when the *event_name* specified as an argument to the on keyword is broadcast.
`send(`*`event_name`*`,`*`state_name`*`)`	none	Send the event_name specified to the state_name specified (directed event broadcasting).
`matlab(evalString,arg1,arg2,...)`	`ml()`	Action specifies a call using MATLAB function notation.
`matlab.MATLAB_workspace_data`	`ml.`	Action specifies a call using the `ml` name space notation.

Use of these keywords in any way other than their intended meaning within the rules of the notation will cause unpredictable results.

How Is Hierarchy Represented?

The notation supports the representation of object hierarchy in Stateflow diagrams. Some of the objects are graphical while others are nongraphical.

An example of a graphical hierarchy is the ability to draw one state within the boundaries of another state. Such a representation indicates that the inner state is a substate or child of the outer state or superstate. The outer state is the parent of the inner state. In the simple case of a Stateflow diagram with a single state, the Stateflow diagram is that state's parent. Transitions are another example of graphical hierarchy. A transition's hierarchy is represented

by determining its parent, source, and destination. In a Stateflow diagram you can see a transition's parent, source, and destination.

Data and event object are nongraphical and their hierarchy is represented differently (using the Explorer) from the graphical object hierarchy (using the graphics editor).

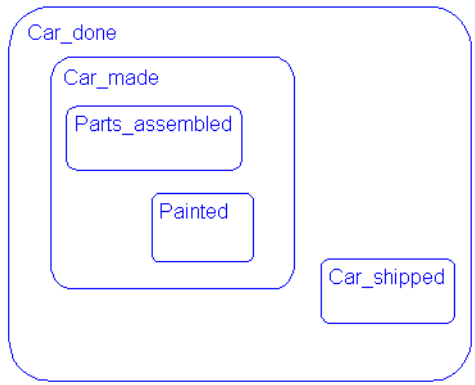
All of the objects in the notation support the representation of hierarchy.

See Chapter 3, "Events and Data," and Chapter 4, "Defining Stateflow Interfaces," for information and examples of representing data and event objects.

For more information on how the hierarchy representations are interpreted, see Chapter 7, "Semantics."

Example: Representing State Hierarchy

This is an example of how state hierarchy is represented.



The Stateflow diagram is the parent of `Car_done`. `Car_done` is the parent state of the `Car_made` and `Car_shipped` states. `Car_made` is also a parent to the `Parts_assembled` and `Car_painted` states. `Parts_assembled` and `Car_painted` are children of the `Car_made` state.

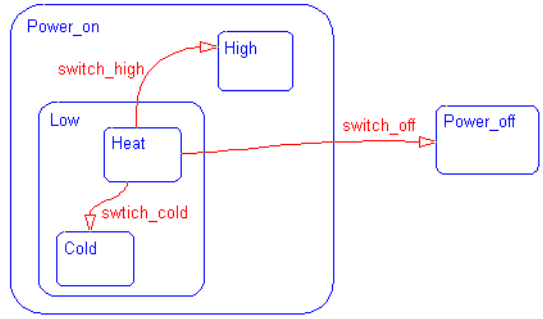
The machine is the root of the Stateflow hierarchy. The Stateflow diagram is represented by the / character. Each level in the hierarchy of states is

separated by the . character. The full hierarchy representation of the state names in this example is:

- /Car_done
- /Car_done.Car_made
- /Car_done.Car_shipped
- /Car_done.Car_made.Parts_assembled
- /Car_done.Car_made.Painted

Example: Representing Transition Hierarchy

This is an example of how transition hierarchy is represented.



A transition's hierarchy is described in terms of the transition's parent, source, and destination. The parent is the lowest level that the transition (source and destination) is contained within. The machine is the root of the hierarchy. The Stateflow diagram is represented by the / character. Each level in the hierarchy of states is separated by the . character. The three transitions in the example are represented as:

Transition Label	Parent	Source	Destination
switch_off	/	/Power_on.Low.Heat	/Power_off
switch_high	/Power_on	/Power_on.Low.Heat	/Power_on.High
switch_cold	/Power_on.Low	/Power_on.Low.Heat	/Power_on.Low.Cold

Example: Representing Event Hierarchy

Event hierarchy is defined by specifying the parent of an event when you create it. Events are nongraphical and are created using either the graphics editor **Add** menu or the Explorer. Using hierarchy you can optimize event processing through directed event broadcasting. Directed event broadcasting is the ability to qualify who can send and receive event broadcasts.

See “Events” on page 3-5 and “Stateflow Explorer User Interface” on page 5-5 for more information.

See “Action Language” on page 6-42 for more information on the notation for directed event broadcasting.

States

Overview

A *state* describes a mode of a reactive system. States in a Stateflow diagram represent these modes. The activity or inactivity of the states dynamically changes based on events and conditions.

Every state has hierarchy. In a Stateflow diagram consisting of a single state, that state's parent is the Stateflow diagram itself. A state also has history that applies to its level of hierarchy in the Stateflow diagram. States can have actions that are executed in a sequence based upon action type. The action types are: `entry`, `during`, `exit`, or `on` *`event_name`* actions.

This table shows the button icon and a short description of a state.

Name	Button Icon	Description
State		Use a state to depict a mode of the system.

Superstate

A state is a superstate if it contains other states, called substates.

Substate

A state is a substate if it is contained by a superstate.

What Is a State Decomposition?

A state has a *decomposition* when it consists of one or more substates. A Stateflow diagram that contains at least one state also has decomposition. Representing hierarchy necessitates some rules around how states can be grouped in the hierarchy. A superstate has either parallel (AND) or exclusive (OR) decomposition. When looking at any one point in the hierarchy, all substates of a superstate must be of the same decomposition.

Parallel (AND) State Decomposition

Parallel (AND) state decomposition is indicated when states have dashed borders. This representation is appropriate if all states at that same level in the hierarchy are active at the same time. The activity within parallel states is essentially independent. The children of parallel (AND) decomposition parents are AND states.

Exclusive (OR) State Decomposition

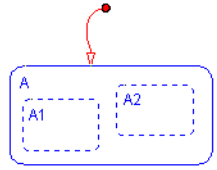
Exclusive (OR) state decomposition is represented by states with solid borders. Exclusive (OR) decomposition is used to describe system modes that are mutually exclusive. When a state has exclusive (OR) decomposition, only one substate can be active at a time. The children of exclusive (OR) decomposition parents are OR states.

Active and Inactive States

States have the Boolean characteristic of being active or inactive. The occurrence of events drives the execution of the Stateflow diagram. At any point in the execution of a Stateflow diagram, there will be some combination of active and inactive states. These are some possible combinations:

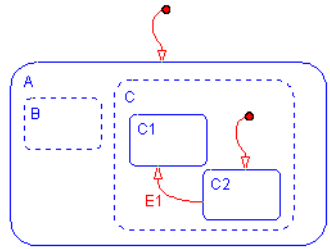
- Multiple active states with parallel (AND) decomposition

  In this example, when state `A` is active, `A1` and `A2` are active.



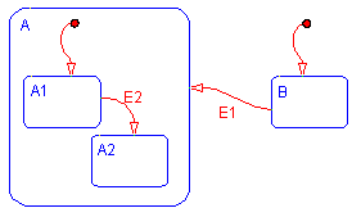
- An active state with parallel (AND) decomposition and an active state with exclusive (OR) decomposition

In this example, state `B` and `C.C2` or state `B` and `C.C1` are active at the same time.



- One active state with exclusive (OR) decomposition

In this example, state `B` or state `A.A1` or state `A.A2` is active at any one time.



When a given state is active, all of its ancestor states are also active. See the section titled “Semantics of Active and Inactive States” on page 7-5 for more information.

What Are Combination States?

When a Stateflow diagram has states with parallel (AND) decomposition, multiple states can be active simultaneously. A combination state is a notational representation of those multiple states. For example, a Stateflow diagram could have two active states with parallel (AND) decomposition, `A.B` and `X.Y`. Using combination state notation, the activity of the Stateflow diagram is denoted by `(A.B,X.Y)`.

A state is characterized by its label. The label consists of the name of the state optionally followed by a / character and additional keywords defined below. The label appears on the top left-hand corner of the state rectangle.

Labeling a State

The ? character is the default state label. State labels have this general format:

```
name/
entry:
during:
exit:
on event_name:
```

The keywords entry (shorthand en), during (shorthand du), exit (shorthand ex), and on identify actions associated with the state. You can specify multiple actions by separating them by any of these:

- Carriage return
- Semicolon
- Comma

Specify multiple on *event_name* actions for different events by adding multiple on *event_name* lines specifying unique values for *event_name*.

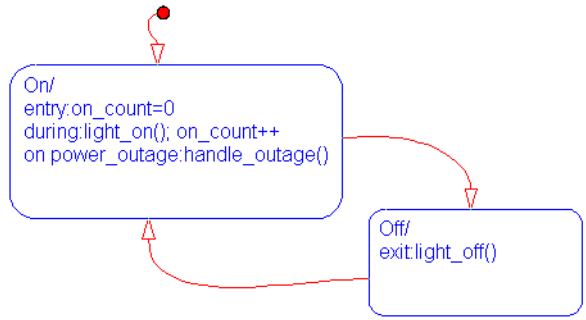
Each keyword is optional and positionally independent. You can specify none, some, or all of them. The colon after each keyword is required. The slash following the state name is optional as long as it is followed by a carriage return.

If you enter the name and slash followed directly by an action or actions (without the entry keyword), the action(s) are interpreted as entry action(s). This shorthand is useful if you are only specifying entry actions.

See the section titled “What Is an Action Language?” on page 6-42 for more information on the action language.

Example: Labeling a State

This example shows the state labeling formats and explains the components of the label.



Name

The *name* of the state forms the first part of the state label. Valid state names consist of alphanumeric characters and can include the _ character, e.g., `Transmission` or `Green_on`.

The use of hierarchy provides some flexibility in the naming of states. The name that you enter as part of the label must be unique when preceded by the hierarchy of its ancestor states. The name as stored in the data dictionary consists of the text you entered as the label on the state, preceded by the hierarchy of its ancestor states separated by periods. States can have the same name appear on the graphical representation of the state, as long as the full names within the data dictionary are unique. The parser indicates an error if a state does not have a unique name entry in the data dictionary for that Stateflow diagram.

See the section titled “Example: Unique State Names” on page 6-15 for an example of uniquely named states.

In this example, the state names are `On` and `Off`.

Entry Action

In the example, state `On` has `entry` action `on_count=0`. The value of `on_count` is reset to 0 whenever state `On`'s `entry` action is executed.

See “Semantics of State Actions” on page 7-7 for information on how and when `entry` actions are executed.

During Action

In the example, state `On` has two `during` actions `light_on()` and `on_count++`. These actions are executed whenever state `On`'s `during` action is executed.

See “Semantics of State Actions” on page 7-7 for information on how and when `during` actions are executed.

Exit Action

In the example, state `Off` has `exit` action `light_off()`. This action is executed whenever state `Off`'s `exit` action is executed.

See “Semantics of State Actions” on page 7-7 for information on how and when `exit` actions are taken.

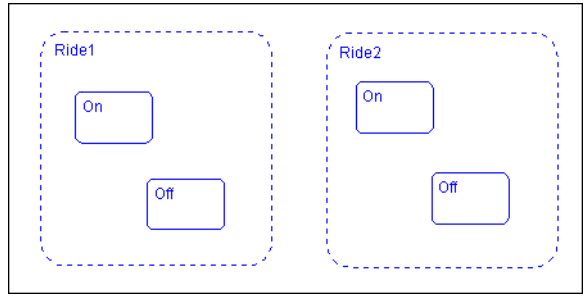
On Event_name Action

In the example, state `Off` has the on *event_name*, `power_outage`. When the event `power_outage` occurs, the action `handle_outage()` is executed.

See “Semantics of State Actions” on page 7-7 for information on how and when on *event_name* actions are taken.

Example: Unique State Names

This example shows how hierarchy supports unique naming of states.



Each of these states has a unique name because of the hierarchy of the Stateflow diagram. Although the name portion of the label on the state itself is not unique, when the hierarchy is prepended to the name in the data

dictionary, the result is unique. The full names for the states as seen in the data dictionary are:

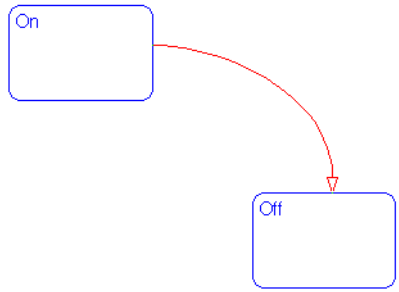
- `Ride1.On`
- `Ride1.Off`
- `Ride2.On`
- `Ride2.Off`

Although the names `On` and `Off` are duplicated, the full names are unique because of the hierarchy of the Stateflow diagram. The example intentionally contains only states for simplification purposes.

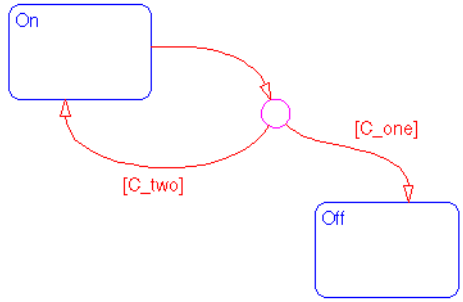
Transitions

What Is a Transition?

In most cases, a *transition* represents the passage of the system from a source object to a destination object. There are transitions between states. There are also transitions between junctions and states. A transition is represented by a line segment ending with an arrow drawn from a source object to the destination object. This is an example of a transition from a source state, `On`, to a destination state, `Off`:



Junctions divide a transition into segments. Each segment is evaluated in the process of determining the validity of the transition from a source to a destination. This is an example of a transition with segments:



A default transition is one special type of transition that has no source object.

Labeling a Transition

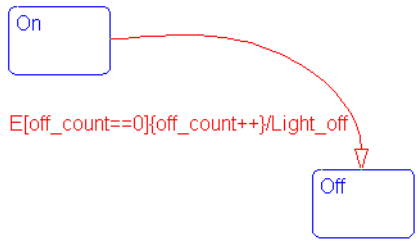
A transition is characterized by its *label*. The label can consist of an event, a condition, a condition action, and/or a transition action. The ? character is the default transition label. Transition labels have this general format:

```
event [condition]{condition_action}/transition_action
```

Replace, as appropriate, your names for event, condition, condition action, and transition action. Each part of the label is optional.

Example: Transition Label

This example shows the format of a transition label.



Event

The specified *event* is what causes the transition to be taken, provided the condition, if specified, is true. Specifying an event is optional. Absence of an event indicates that the transition is taken upon the occurrence of any event. Multiple events are specified using logical operators (&, |).

In this example, the broadcast of event `E`, triggers the transition from `On` to `Off`, provided the condition, `[off_count==0]`, is true.

Condition

A *condition* is a Boolean expression to specify that a transition occurs given that the specified expression is true. Enclose the condition in square brackets. See the section titled “Conditions” on page 6-61 for information on the condition notation.

In this example, the condition `[off_count==0]` must evaluate as true for the condition action to be executed and for transition from the source to the destination to be valid.

Condition Action

The *condition action* is executed as soon as the condition, if specified, is evaluated as true and before the transition destination has been determined to be valid.

If the transition consists of multiple segments, the condition action is executed as soon as the condition, if specified, is evaluated as true and before the transition segment is determined as valid. Enclose the condition action in curly brackets. See the section titled “Action Language” on page 6-42 for more information on the action language.

If no condition is specified, the implied condition is always evaluated as true.

In this example, if the condition `[off_count==0]` is true, the condition action, `off_count++` is immediately executed.

Transition Action

The *transition action* is executed after the transition destination has been determined to be valid provided the condition, if specified, is true. If the transition consists of multiple segments, the transition action is only executed when the entire transition path to the final destination is determined as valid. Precede the transition action with a backslash. See the section titled “Action Language” on page 6-42 for more information on the action language.

In this example, if the condition `[off_count==0]` is true, and the destination state `Off` is valid, the transition action `Light_off` is executed.

What Is a Valid Transition?

In most cases, a transition is valid when the source state of the transition is active and the transition label is valid. Default transitions are slightly different because there is no source state. Validity of a default transition to a substate is evaluated when there is a transition to its superstate assuming the superstate

is active. This labeling criteria applies to both default transitions and general case transitions. These are possible combinations of valid transition labels:

Transition Label	Is Valid If:
Event only	That event occurs
Event and condition	That event occurs and the condition is true
Condition only	Any event occurs and the condition is true
Action only	Any event occurs
Not specified	Any event occurs

Types of Transitions

The notation supports these transition types:

- Transitions to and from exclusive (OR) states

  See the sections titled “Example: Transitions to and from Exclusive (OR) States” on page 6-21 for an example of this type of transition.

- Transitions to and from junctions

  See the section titled “Example: Transitions to and from Junctions” on page 6-21 for an example of this type of transition.

- Transitions to and from exclusive (OR) superstates

  See the section titled “Example: Transitions to and from Exclusive OR Superstates” on page 6-22 for an example of this type of transition.

- Transitions from no source to an exclusive (OR) state (default transitions)

  See the section titled “What is a Default Transition?” on page 6-24 for examples of this type of transition.

- Inner state transitions

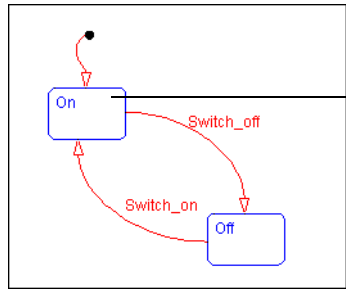
  See the section titled “What Is an Inner Transition?” on page 6-27 for examples of this type of transition.

- Self loop transitions

  See the section titled “What Is a Self Loop Transition?” on page 6-31 for examples of this type of transition.

Example: Transitions to and from Exclusive (OR) States

This example shows simple transitions to and from exclusive (OR) states.

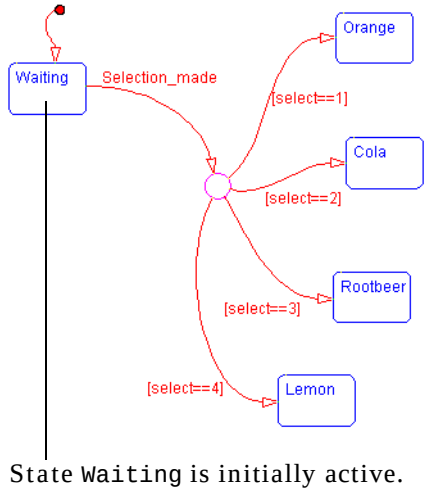


The transition `On→Off` is valid when state `On` is active and the event `Switch_off` occurs. The transition `Off→On` is valid when state `Off` is active and event `Switch_on` occurs.

See the section titled “Transitions to and from Exclusive (OR) States” on page 7-8 for more information on the semantics of this notation.

Example: Transitions to and from Junctions

This example shows transitions to and from a connective junction.

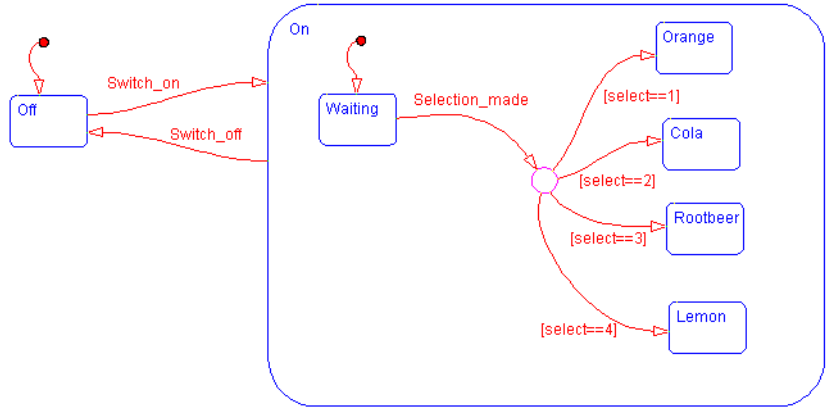


This is a Stateflow diagram of a soda machine. The Stateflow diagram is called when the external event `Selection_made` occurs. The Stateflow diagram awakens with the `Waiting` state active. The `Waiting` state is a common source state. When the event `Selection_made` occurs, the Stateflow diagram transitions from the `Waiting` state to one of the other states based on the value of the variable select. One transition is drawn from the `Waiting` state to the connective junction. Four additional transitions are drawn from the connective junction to the four possible destination states.

See the section titled “Example: Transitions from a Common Source to Multiple Destinations” on page 7-39 for more information on the semantics of this notation.

Example: Transitions to and from Exclusive OR Superstates

This example shows transitions to and from an exclusive (OR) superstate and the use of a default transition.



This is an expansion of the soda machine Stateflow diagram that includes the initial example of the `On` and `Off` exclusive (OR) states. On is now a superstate containing the `Waiting` and soda choices states. The transition `Off`→`On` is valid when state `Off` is active and event `Switch_on` occurs. Now that `On` is a superstate, this is an explicit transition to the `On` superstate.

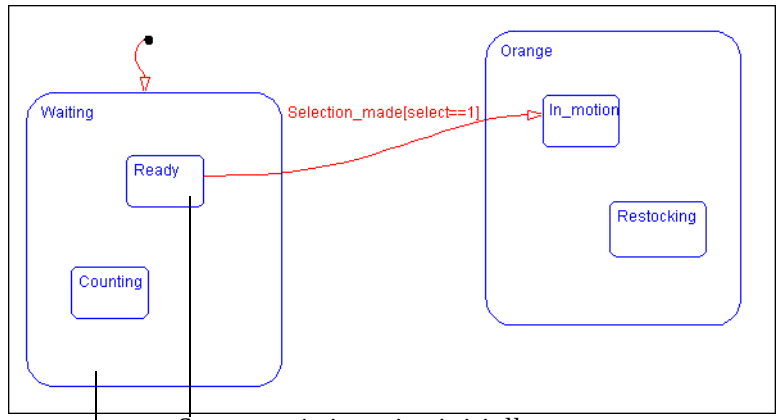
To be a valid transition to a superstate, the destination substate must be implicitly defined. By defining that the `Waiting` substate has a default transition, the destination substate is implicitly defined. This notation defines that the resultant transition is `Off`→ `On.Waiting`.

The transition `On`→`Off` is valid when state `On` is active and event `Switch_off` occurs. When the `Switch_off` event occurs, no matter which of the substates of `On` is active, we want to transition to the `Off` state. This top-down approach supports the ability to simplify the Stateflow diagram by looking at the transitions out of the superstate without considering all the details of states and transitions within the superstate.

See the section titled “Default Transitions” on page 7-18 for more information on the semantics of this notation.

Example: Transitions to and from Substates

This example shows transitions to and from exclusive (OR) substates.



Two of the substates of the `On` superstate are further defined to be superstates of their own. The Stateflow diagram shows a transition from one OR substate to another OR substate. The transition `Waiting.Ready`→`Orange.In_motion` is valid when state `Waiting.Ready` is active and event `Selection_made` occurs, providing that the select variable equals one. This transition defines an explicit exit from the `Waiting.Ready` state and an implicit exit from the `Waiting` superstate. On the destination side, this transition defines an implicit entry

into the `Orange` superstate and an explicit entry into the `Orange.In_motion` substate.

See the section titled “Example: Transition from a Substate to a Substate” on page 7-11 for more information on the semantics of this notation.

What is a Default Transition?

Default transitions are primarily used to specify which exclusive (OR) state is to be entered when there is ambiguity among two or more neighboring exclusive (OR) states. For example, default transitions specify which substate of a superstate with exclusive (OR) decomposition the system enters by default in the absence of any other information such as a history junction. Default transitions are also used to specify that a junction should be entered by default. The default transition object is a transition with a destination but no source object.

Click on the **Default transition** button in the toolbar, and click on a location in the drawing area close to the state or junction you want to be the destination for the default transition. Drag the mouse to the destination object to attach the default transition. In some cases it is useful to label default transitions.

One of the most common Stateflow programming mistakes is to create multiple exclusive (OR) states without a default transition. In the absence of the default transition, there is no indication of which state becomes active by default. Note that this error is flagged when you simulate the model using the Debugger with the **State Inconsistencies** option enabled.

This table shows the button icon and briefly describes a default transition.

Name	Button Icon	Description
Default transition		Use a Default transition to indicate, when entering this level in the hierarchy, which object becomes active by default.

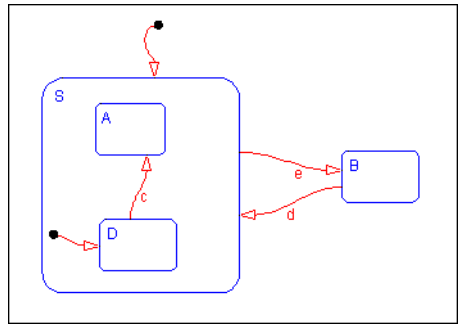
Labeling of Default Transitions

In some circumstances, you may want to label default transitions. You can label default transitions as you would other transitions. For example, you may want to specify that one state or another should become active depending upon

the event that has occurred. In another situation, you may want to have specific actions take place that are dependent upon the destination of the transition.

Example: Use of Default Transitions

This example shows a use of default transitions.



When the Stateflow diagram is first awakened, the default transition to superstate S defines that of states S and B; the transition to state S is valid. State S has two substates, A and D. Which substate does the system transfer to? It cannot transfer to both of them since A and D are not parallel (AND) states. Again, this kind of ambiguity is cleared up by defining a default transition to substate D.

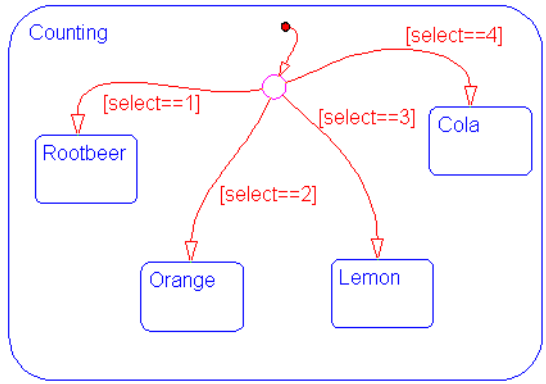
Suppose at a different execution point, the Stateflow diagram is awakened by the occurrence of event d and state B is active. The transition B→S is valid. When the system enters state S, it enters substate D because the default transition is defined.

See the section titled “Default Transitions” on page 7-18 for more information on the semantics of this notation.

The default transitions are required for the Stateflow diagram to execute. Without the default transition to state S, when the Stateflow diagram is awakened, none of the states become active. You can detect this situation at runtime by checking for state inconsistencies. See “Dynamic Checker Options” on page 10-10 for more information.

Example: Default Transition to a Junction

This example shows a default transition to a connective junction.

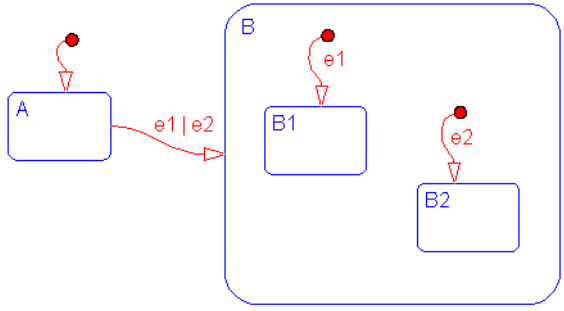


In this example, the default transition to the connective junction defines that upon entering the `Counting` state, the destination is determined by the condition on each transition segment.

See the section titled “Example: Default Transition to a Junction” on page 7-20 for more information on the semantics of this notation.

Example: Default Transition with a Label

This example shows a use of labeling default transitions.



If state A is initially active and either e1 or e2 occurs, the transition from state A to superstate B is valid. The substates B1 and B2 both have default transitions. The default transitions are labeled to specify the event that triggers the transition. If event e1 occurs, the transition A→B1 is valid. If event e2 occurs, the transition A→B2 is valid.

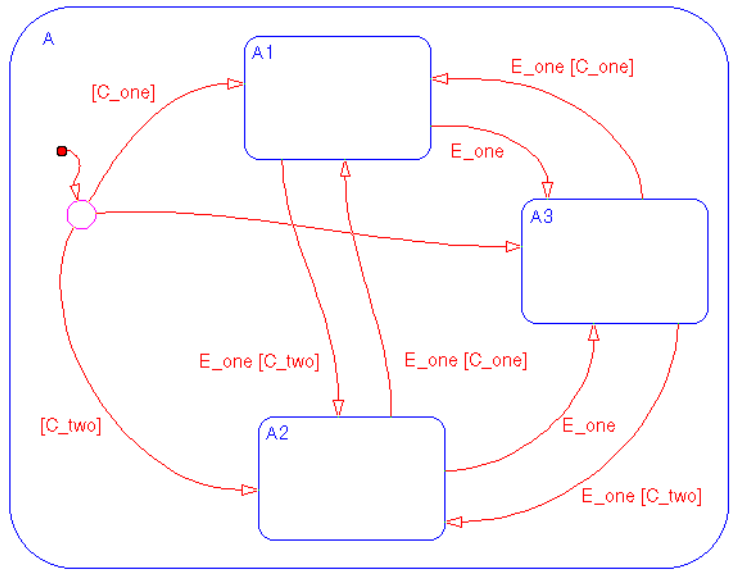
See the section titled “Example: Labeled Default Transitions” on page 7-23 for more information on the semantics of this notation.

What Is an Inner Transition?

An *inner transition* is a transition that does not exit the source state. Inner transitions are most powerful when defined for superstates with exclusive (OR) decomposition. Use of inner transitions can greatly simplify a Stateflow diagram.

Example One: Before Using an Inner Transition

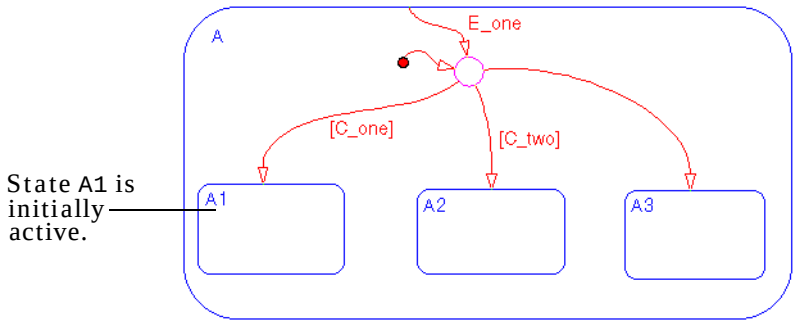
This is an example of a Stateflow diagram that could be simplified by using an inner transition.



Any event occurs and awakens the Stateflow diagram. The default transition to the connective junction is valid. The destination of the transition is determined by [C_one] and [C_two]. If [C_one] is true, the transition to A1 is true. If [C_two] is true, the transition to A2 is valid. If neither [C_one] nor [C_two] is true, the transition to A3 is valid. The transitions among A1, A2, and A3 are determined by E_one, [C_one], and [C_two].

Example One: Inner Transition to a Connective Junction

This example shows a solution to the same problem (Example One) using an inner transition to a connective junction.



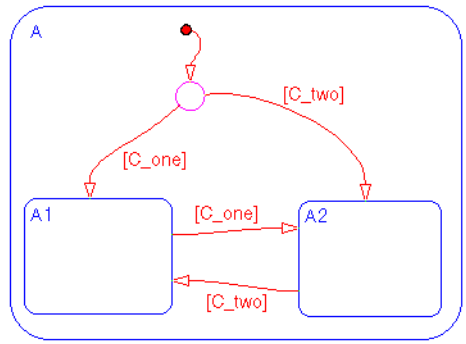
Any event occurs and awakens the Stateflow diagram. The default transition to the connective junction is valid. The destination of the transitions is determined by [C_one] and [C_two].

The Stateflow diagram is simplified by using an inner transition in place of the many transitions amongst all the states in the original example. If state A is already active, the inner transition is used to re-evaluate which of the substates of state A is to be active. When event E_one occurs, the inner transition is potentially valid. If [C_one] is true, the transition to A1 is valid. If [C_two] is true, the transition to A2 is valid. If neither [C_one] nor [C_two] is true, the transition to A3 is valid. This solution is much simpler than the previous one.

See the section titled “Example: Processing One Event with an Inner Transition to a Connective Junction” on page 7-28 for more information on the semantics of this notation.

Example Two: Before Using an Inner Transition

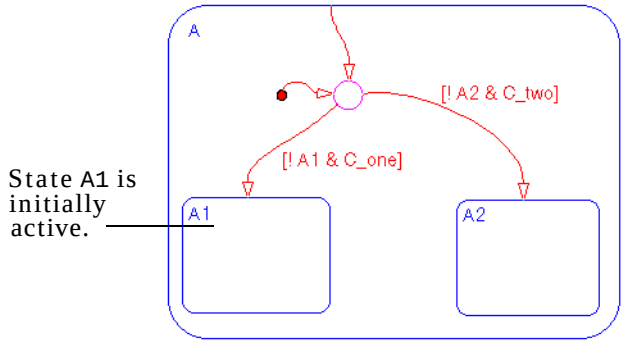
This is an example of a Stateflow diagram that could be simplified by using an inner transition.



Any event occurs and awakens the Stateflow diagram. The default transition to the connective junction is valid. The destination of the transition is determined by the conditions `[C_one]` and `[C_two]`. If `[C_one]` is true, the transition to `A1` is true. If `[C_two]` is true, the transition to `A2` is valid. The transitions between `A1` and `A2` are also determined by `[C_one]` and `[C_two]`.

Example Two: Inner Transition to a Connective Junction

This example shows a solution to the same problem (Example Two) using an inner transition to a connective junction.



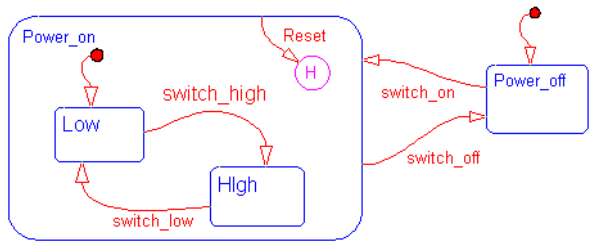
Any event occurs and awakens the Stateflow diagram. The default transition to the connective junction is valid. The destination of the transitions is determined by [C_one] and [C_two]. (When the default transition path is followed, neither state A1 nor A2 is active; thus the truth of condition is based exclusively on [C_one] and [C_two].)

The Stateflow diagram is simplified by using an inner transition in place of the transitions between A.A1 and A.A2 in the original example. If state A is already active, the inner transition is used to re-evaluate which of the substates of state A is to be active. If state A1 is not active and [C_one] is true, the transition to A1 is valid. If state A2 is not active and [C_two] is true, the transition to A2 is valid. This solution is much simpler than the previous one.

See the section titled "Example: Processing One Event with an Inner Transition to a Connective Junction" on page 7-28 for more information on the semantics of this notation.

Example: Inner Transition to a History Junction

This example shows an inner transition to a history junction.



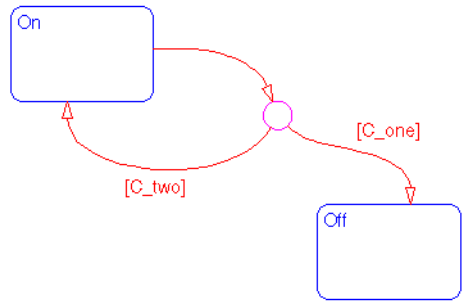
State Power_on.High is initially active. When event Reset occurs, the inner transition to the history junction is valid. Because the inner transition is valid, the currently active state, Power_on.High, will be exited. When the inner transition to the history junction is processed, the last active state, Power_on.High, becomes active (is re-entered). If Power_on.Low was active under the same circumstances, Power_on.Low would be exited and re-entered as a result. The inner transition in this example is equivalent to drawing an outer self-loop transition on both Power_on.Low and Power_on.High.

See the section titled "Example: Use of History Junctions" on page 6-40 for another example using a history junction.

See the section titled “Example: Inner Transition to a History Junction” on page 7-31 for more information on the semantics of this notation.

What Is a Self Loop Transition?

A transition segment from a state to a connective junction that has an outgoing transition segment from the connective junction back to itself is a self loop. This is an example of a self loop:



See these sections for examples of self loops:

- “Example: Connective Junction Special Case - Self Loop” on page 6-34

  See the section titled “Example: Self Loop” on page 7-34 for information on the semantics of this notation.
- “Example: Connective Junction and For Loops” on page 6-35

  See the section titled “Example: For Loop Construct” on page 7-35 for information on the semantics of this notation.

Connective Junctions

What Is a Connective Junction?

A connective junction is used to represent a decision point in the Stateflow diagram. The connective junction enables representation of different transition paths. Connective junctions are used to help represent:

- Variations of an `if-then-else` decision construct by specifying conditions on some or all of the outgoing transitions from the connective junction.
- A self loop back to the source state if none of the outgoing transitions is valid.
- Variations of a `for` loop construct by having a self loop transition from the connective junction back to itself.
- Transitions from a common source to muliple destinations.
- Transitions from multiple sources to a common destination.
- Transitions from a source to a destination based on common events

See the section titled “Event Actions” on page 7-43 for a summary of the semantics of connective junctions.

What Is Flow Diagram Notation?

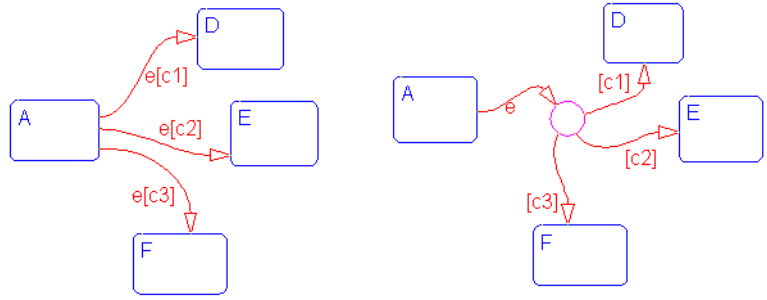
Flow diagram notation is essentially logic represented without the use of states. In some cases, using flow diagram notation is a closer representation of the system's logic and avoids the use of unnecessary states. Flow diagram notation is an effective way to represent common code structures like `for` loops and `if-then-else` constructs. The use of flow diagram notation in a Stateflow diagram can produce more efficient code optimized for memory use. Reducing the number of states optimizes memory use.

Flow diagram notation is represented through combinations of self-loops to connective junctions, transitions to and from connective junctions, and inner transitions to connective junctions. The key to representing flow diagram notation is in the labeling of the transitions (specifically the use of action language).

Flow diagram notation and state-to-state transition notation seamlessly coexist in the same Stateflow diagram.

Example: Connective Junction with All Conditions Specified

When event e occurs, state A transfers to D, E, or F depending on which of the conditions [c1], [c2], or [c3] is met. With the alternative representation, using a connective junction, the transition from A to the connective junction occurs first, provided the event has occurred. A destination state is then determined based on which of the conditions [c1], [c2], or [c3] is satisfied. The transition from the source state to the connective junction is labeled by the event, and those from the connective junction to the destination states by the conditions. No event is applicable in a transition from a connective junction to a destination state.

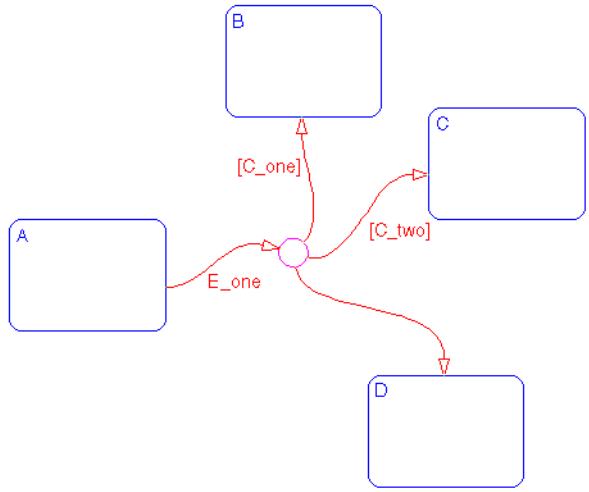


See the section titled “Example: If-Then-Else Decision Construct” on page 7-33 for information on the semantics of this notation.

Example: Connective Junction with One Unconditional Transition

The transition A→B is valid when A is active, event E_one occurs, and [C_one] is true. The transition A→C is valid when A is active, event E_one occurs, and [C_two] is true. Otherwise, given A is active and event E_one occurs, the

transition A→D is valid. If you do not explicitly specify condition [C_three], it is implicit that the transition condition is not [C_one] and not [C_two].



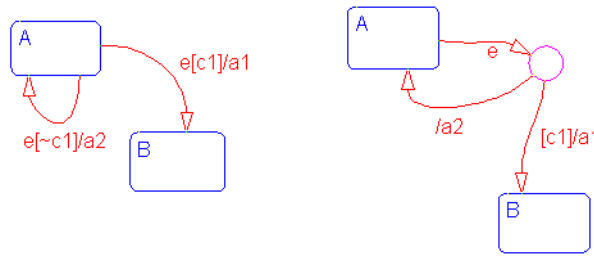
See the section titled “Example: If-Then-Else Decision Construct” on page 7-33 for information on the semantics of this notation.

Example: Connective Junction Special Case - Self Loop

In some situations, the transition event occurs, but the condition is not met. The transition cannot be taken, but an action is generated. You can represent this situation by using a connective junction or a self loop (transition from state to itself).

In state A, event e occurs. If condition [c1] is met, transition A→B is taken, generating action a1. The transition A→A is valid if event e occurs and [c1] is not true. In this self loop, the system exits and re-enters state A, and executes action a2. An alternative representation using a connective junction is shown.

The two representations are equivalent; in the one that uses a connective junction, it is not necessary to specify condition `[~c1]` explicitly, as it is implied.



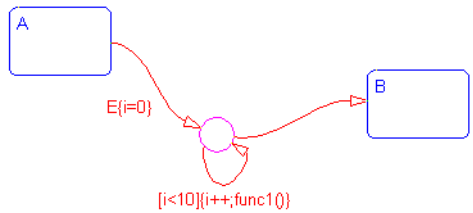
See the section titled “Example: Self Loop” on page 7-34 for information on the semantics of this notation.

Example: Connective Junction and For Loops

This example shows a combination of flow diagram notation and state transition notation. Self loops to connective junctions can be used to represent `for` loop constructs.

In state `A`, event `E` occurs. The transition from state `A` to state `B` is valid if the conditions along the transition path are true. The first segment of the transition does not have a condition, but does have a condition action. The condition action, `{i=0}`, is executed. The condition on the self loop is evaluated as true and the condition actions `{i++;func1()}` execute. The condition actions execute until the condition, `[i<10]`, is false. The condition actions on both the first segment and the self loop to the connective junction effectively execute a `for` loop (for `i` values 0 to 9 execute `func1()`). The `for` loop is executed outside of the context of a state. The remainder of the path is

evaluated. Since there are no conditions, the transition completes at the destination, state B.



See the section titled “Example: For Loop Construct” on page 7-35 for information on the semantics of this notation.

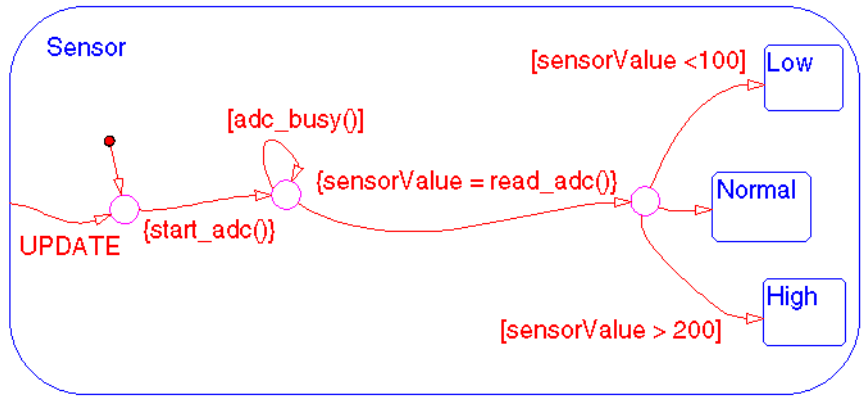
Example: Flow Diagram Notation

This example shows a real-world use of flow diagram notation and state transition notation. This Stateflow diagram models an 8-bit analog-to-digital converter (ADC).

Consider the case when state `Sensor.Low` is active and event `UPDATE` occurs. The inner transition from `Sensor` to the connective junction is valid. The next transition segment has a condition action, `{start_adc()}`, which initiates a reading from the ADC. The self-loop on the second connective junction repeatedly tests the condition `[adc_busy()]`. This condition evaluates as true once the reading settles (stabilizes) and the loop completes. This self loop is used to introduce the delay needed for the ADC reading to settle. The delay could have been represented by using another state with some sort of counter. Using flow notation in this example avoids an unnecessary use of a state and produces more efficient code.

The next transition segment condition action, `{sensorValue=read_adc()}`, puts the new value read from the ADC in the data object `sensorValue`. The final transition segment is determined by the value of `sensorValue`. If `[sensorValue <100]` is true, the state `Sensor.Low` is the destination. If

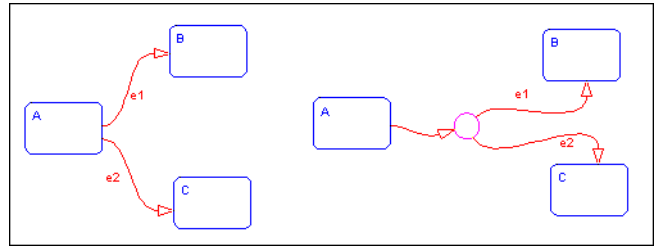
[sensorValue >200] is true, the state Sensor.High is the destination. Otherwise, state Sensor.Normal is the destination state.



See the section titled “Example: Flow Diagram Notation” on page 7-37 for information on the semantics of this notation.

Example: Connective Junction from a Common Source to Multiple Destinations

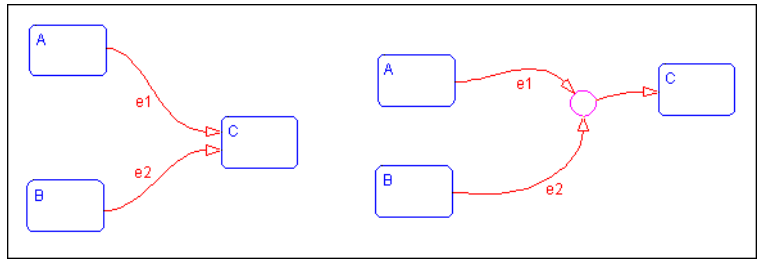
Transitions A→B and A→C share a common source state A. An alternative representation uses one arrow from A to a connective junction, and multiple arrows labeled by events from the junction to the destination states B and C.



See the section titled “Example: Transitions from a Common Source to Multiple Destinations” on page 7-39 for information on the semantics of this notation.

Example: Connective Junction Common Destination

Transitions A→C and B→C share a common destination state C. An alternative representation uses one transition from A to a connective junction, one transition from B to the same connective junction, and one transition from the connective junction to the destination state, C.



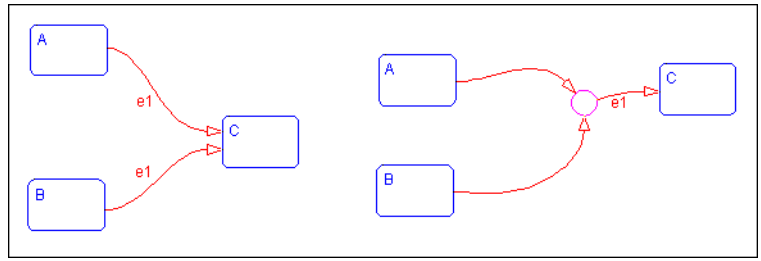
See the section titled “Example: Transitions from Multiple Sources to a Common Destination” on page 7-40 for information on the semantics of this notation.

Example: Connective Junction Common Events

Suppose, for example, that when event e1 occurs, the system, whether it is in state A or B, will transfer to state C. Suppose that transitions A→C and B→C are triggered by the same event e1, so that both destination state and trigger event are common between the transitions. There are three ways to represent this:

- By drawing transitions from A and B to C, each labeled with e1
- By placing A and B in one superstate S, and drawing one transition from S to C, labeled with e1
- By drawing transitions from A and B to a connective junction, then drawing one transition from the junction to C, labeled with e1

This Stateflow diagram shows the simplification using a connective junction.



See the section titled “Example: Transitions from a Source to a Destination Based on a Common Event” on page 7-41 for information on the semantics of this notation.

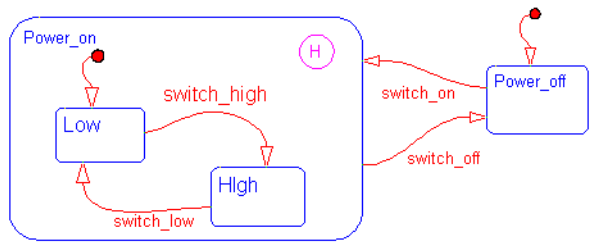
History

What Is a History Junction?

A history junction is used to represent historical decision points in the Stateflow diagram. The decision points are based on historical data relative to state activity. Placing a history junction in a superstate indicates that historical state activity information is used to determine the next state to become active. The history junction applies only to the level of the hierarchy in which it appears.

Example: Use of History Junctions

This example shows a use of history junctions:



Superstate `Power_on` has a history junction and contains two substates. If state `Power_off` is active and event `switch_on` occurs, the system could enter either `Power_on.Low` or `Power_on.High`. The first time superstate `Power_on` is entered, substate `Power_on.Low` will be entered because it has a default transition. At some point afterwards, if state `Power_on.High` is active and event `switch_off` occurs, superstate `Power_on` is exited and state `Power_off` becomes active. Then event `switch_on` occurs. Since `Power_on.High` was the last active state, it becomes active again. After the first time `Power_on` becomes active, the choice between entering `Power_on.Low` or `Power_on.High` is determined by the history junction.

See the section titled “Example: Default Transition and a History Junction” on page 7-21 for more information on the semantics of this notation.

History Junctions and Inner Transitions

By specifying an inner transition to a history junction, you can specify that, based on a specified event and/or condition, the active state is to be exited and then immediately re-entered.

See the section titled “Example: Inner Transition to a History Junction” on page 6-30 for an example of this notation.

See the section titled “Example: Inner Transition to a History Junction” on page 7-31 for more information on the semantics of this notation.

Action Language

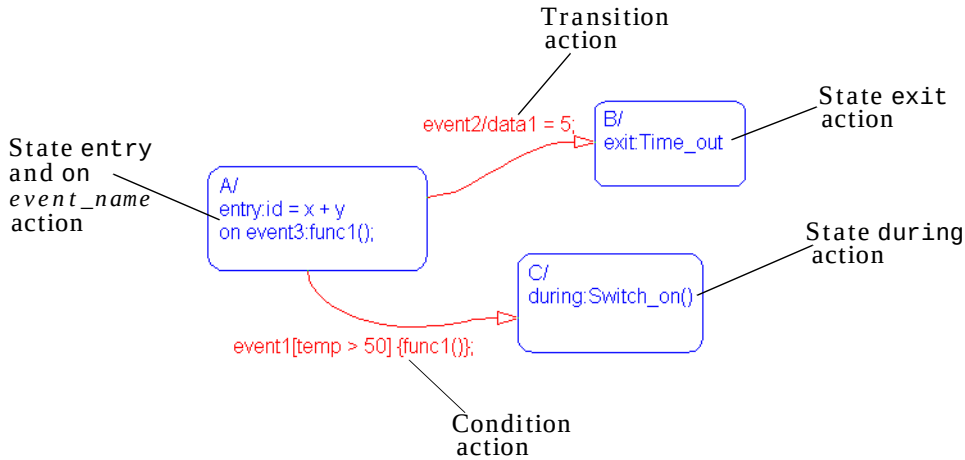
What Is an Action Language?

You sometimes want actions to take place as part of Stateflow diagram execution. The action can be executed as part of a transition from one state to another, or it can depend on the activity status of a state. Transitions can have condition actions and transition actions. States can have `entry`, `during`, `exit`, and, on *`event_name`* actions.

An action can be a function call, an event to be broadcast, a variable to be assigned a value, etc. The *action language* defines the categories of actions you can specify and their associated notations. Violations of the action language notation are flagged as errors by the parser. This section describes the action language notation rules.

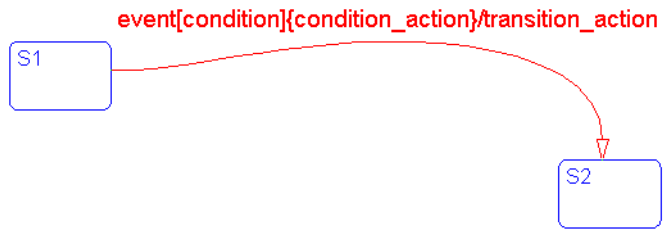
Objects with Actions

This Stateflow diagram shows examples of the possible transition and state actions.



Transition Action Notation

Actions can be associated with transitions via the transition's label. The general format of a transition label is:



When the event occurs, the transition is evaluated. The condition action is executed as soon as the condition is evaluated as true and before the transition destination has been determined to be valid. Enclose the condition action in curly brackets. Specifying a transition action means that the action is executed when the transition is taken, provided the condition, if specified, is true.

State Action Notation

Actions can be associated with states via the state's label by defining `entry`, `during`, `exit`, and on *`event_name`* keywords. The general format of a state label is:

See the section titled "Semantics of State Actions" on page 7-7 for information on the semantics of state actions. See the examples in Chapter 7, "Semantics," for examples of the semantics of state actions.

Action Language Components

The action language defines usage rules for these components:

- Bit operations
- Binary operations
- Unary operations
- Assignment operations
- User-written function calls
- `ml()` function
- `ml.` name space operator
- Data and event arguments
- Arrays
- Event broadcasting
- Directed event broadcasting
- Conditions
- Time, t
- Literals
- Continuation symbols
- Comments
- Semicolon

Bit Operations

You can enable C-like bit operations by adding the `bitops` flag to the code generation command in the **Coder Command** field. See “Make Command Customization” on page 9-23 and “Compiling Code/Building Targets” on page 9-26 for more information on the **Coder Command** field. If you have `bitops` enabled, some of the logical binary operators and unary operators are interpreted as bitwise operators. See “Binary Operations” on page 6-45 and “Unary Operations” on page 6-47 for specific interpretations.

Binary Operations

Binary operations fall into these categories:

- Numerical

Example	Description
`a + b`	Addition of two operands
`a - b`	Subtraction of one operand from the other
`a * b`	Multiplication of two operands
`a / b`	Division of one operand by the other
`a %% b`	Modulus

- Logical (the default setting; bit operations not enabled)

Example	Description
`a == b`	Comparison of equality of two operands
`a & b` `a && b`	Logical AND of two operands

Example	Description
`a | b` `a || b`	Logical OR of two operands
`a ~= b` `a != b`	Comparison of inequality of two operands
`a > b`	Comparison of the first operand greater than the second operand
`a < b`	Comparison of the first operand less than the second operand
`a >= b`	Comparison of the first operand greater than or equal to the second operand
`a <= b`	Comparison of the first operand less than or equal to the second operand

- Logical (bit operations enabled)

Example	Description
`a == b`	Comparison of equality of two operands
`a && b`	Logical AND of two operands
`a & b`	Bitwise AND of two operands
`a || b`	Logical OR of two operands
`a | b`	Bitwise OR of two operands

Example	Description

`a ~= b`
`a != b`
`a <> b`
	Comparison of inequality of two operands

`a > b`	Comparison of the first operand greater than the second operand
`a < b`	Comparison of the first operand less than the second operand
`a >= b`	Comparison of the first operand greater than or equal to the second operand
`a <= b`	Comparison of the first operand less than or equal to the second operand
`a ^ b`	Bitwise XOR of two operands

Unary Operations

These unary operations are supported: ~, !, -.

Example	Description
`~a`	Logical not of `a` Complement of `a` (if `bitops` is enabled)
`!a`	Logical not of `a`
`-a`	Negative of `a`

Unary Actions

These unary actions are supported:

Example	Description
`a++`	Increment a
`a--`	Decrement a

Assignment Operations

These assignment operations are supported:

Example	Description
`a = expression`	Simple assignment
`a += expression`	Equivalent to `a = a + expression`
`a -= expression`	Equivalent to `a = a - expression`
`a *= expression`	Equivalent to `a = a * expression`
`a /= expression`	Equivalent to `a = a / expression`

These additional assignment operations are supported when bit operations are enabled:

Example	Description
`a |= expression`	Equivalent to `a = a | expression` (bit operation)
`a &= expression`	Equivalent to `a = a & expression` (bit operation)
`a ^= expression`	Equivalent to `a = a ^ expression` (bit operation)

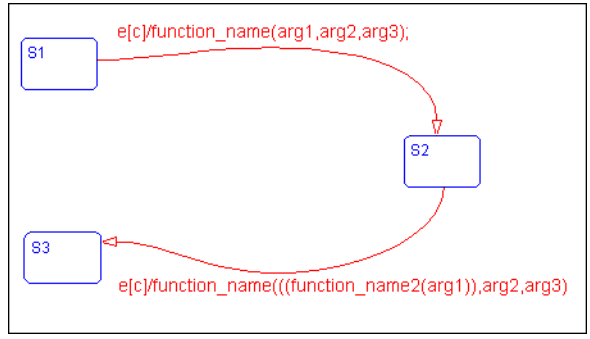
User-Written Functions

You can specify calls to user-written functions in the action language. These guidelines apply to user-written function calls.

- Define a function by its name, any arguments in parenthesis, and an optional semicolon.
- You can specify up to 50 arguments in the function call.
- String parameters to user-written functions are passed between single quotes. For example, `func('string')`.
- You can nest function calls.

Example: Function Call Transition Action

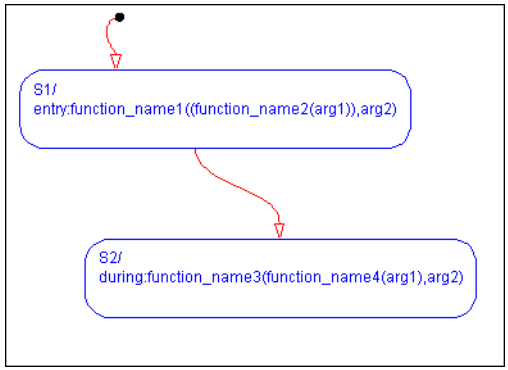
These are example formats of function calls using transition action notation.



If `S1` is active, event `e` occurs, `c` is true, and the transition destination is determined, then a function call is made to `function_name` with `arg1`, `arg2`, and `arg3`. The transition action in the transition from `S2` to `S3` shows a function call nested within another function call.

Example: Function Call State Action

These are example formats of function calls using state action notation.



When the default transition into `S1` occurs, `S1` is marked active and then its `entry` action, a function call to `function_name1` with the specified arguments, is executed and completed. If `S2` is active and an event occurs, the `during` action, a function call to `function_name3` with the specified arguments, executes and completes.

ml() Functions

You can specify calls to MATLAB functions that return scalars (of type double) in the action language.

ml() Function Format

The format of the ml() function is:

```
ml(evalString, arg1, arg2, arg3,...);
```

where the return value is scalar (of type double).

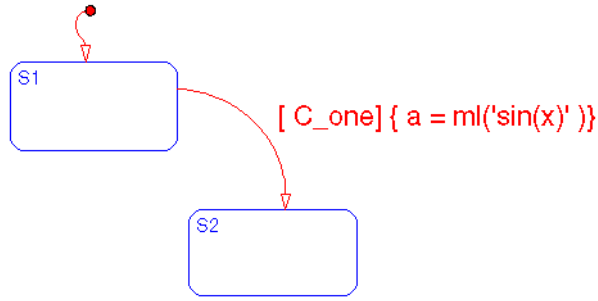
If the result returned is:

- A vector, then the first element is returned.
- A void, then an appropriate format must be used (an assignment statement cannot be used).
- A string, a structure, or a cell array, then the behavior is undefined.

`evalString` is a string that is evaluated in the MATLAB workspace with formatted substitutions of `arg1`, `arg2`, `arg3`, etc.

Example One: ml() Function Call

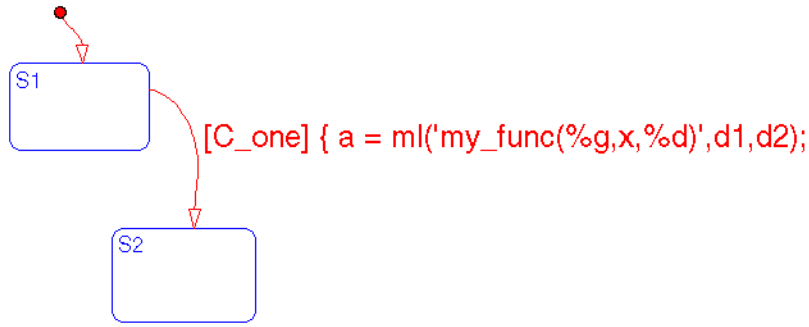
This is an example of an `ml()` function call as part of a condition action.



If `S1` is active, an event occurs, and if `[c_one]` is true, the expression `sin(x)` is evaluated in the MATLAB workspace and the return value assigned to `a`. (`x` must be a variable in the MATLAB workspace and `a` is a data object in the Stateflow diagram). The result of the evaluation must be a scalar. If `x` is not defined in the MATLAB workspace, a runtime error is generated.

Example Two: ml() Function Call

This is an example of a `ml()` function call that passes Stateflow data as arguments. Notice the use of format specifiers `%g` and `%d` as are used in the C language function `printf`.



These data objects are defined:

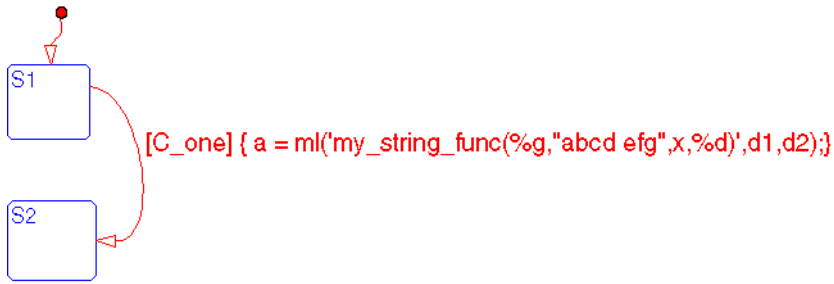
- d1 and a are **Local** data objects of type double in the Stateflow diagram
- d2 is an **Output to Simulink** data object of type integer in the Stateflow diagram
- x must be defined in the MATLAB workspace prior to the execution of the condition action where it is used; if it is not defined, a runtime error is generated.

These three values are passed as arguments to a user-written function. The %g and %d characters are format specifiers that print the current values of d1 and d2 into evalString at appropriate locations.

For example if d1 equals 3.4 and d2 equals 5, using the format specifiers these are mapped into my_func(3.4,x,5). This string is then sent to MATLAB and is executed in the MATLAB workspace.

Example Three: ml() Function Call

This is an example of a ml() function call with string arguments.



These data objects are defined in the Stateflow diagram:

- d1 is a **Local** data object of type double
- d2 is an **Output to Simulink** data object of type integer

The user-written function my_string_func expects four arguments, where the second argument is a string. The %g and %d characters are format specifiers that print the current values of d1 and d2 into evalString at appropriate locations. Notice that the string is enclosed in two single quotes.

Use Guidelines

These guidelines apply to `ml()` functions:

- The first argument must be a string.
- If there are multiple arguments, ensure that the number and types of format specifiers (%g, %d, etc.) match the actual number and types of the arguments. These format specifiers are the same as those used in the C function `printf`.
- A scalar (of type double) is returned.
- ml() function calls can be nested.
- Calls to `ml()` functions should be avoided if you plan to build an RTW target that includes code from Stateflow Coder.

ml. Name Space Operator

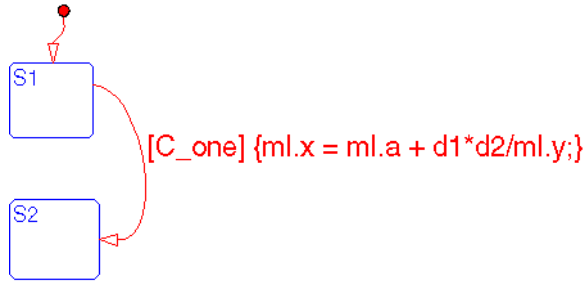
The MATLAB name space operator, `ml`, is used to get and set variables in the MATLAB workspace. The `ml` operator can also be used to access MATLAB functions that operate on scalars in a convenient format.

Use the notation, `a = ml.func_name();`, to call a MATLAB function that does not accept any arguments. Omission of the empty brackets causes a search for a variable of the name specified. The variable will not be found and a runtime error is encountered during simulation.

Use of the ml. name space operator should be avoided if you plan to build an RTW target that includes code from Stateflow Coder.

Example: Using the ml Operator to Access MATLAB Workspace Variables

This is an example of using the `ml` operator to get and set variables in the MATLAB workspace:



These data objects are defined in the Stateflow diagram:

- `d1` and `d2` are **Local** data objects
- `a`, `x`, and `y` must be defined in the MATLAB workspace prior to starting the simulation; otherwise a runtime error is generated at the execution time of the transition

The values of `a` and `y` are accessed in the MATLAB workspace and used in the expression with the **Local** data objects `d1` and `d2`. The result of the expression is assigned to the MATLAB workspace variable `x`. If `x` does not exist, it is automatically created in the MATLAB workspace.

Example: Using the ml Operator to Access MATLAB Functions

This is an example of using the `ml` operator to access MATLAB functions:

These data objects are defined:

- d1 and d2 are **Local** data objects defined in the Stateflow diagram
- x is assumed to be a two dimensional array in the MATLAB workspace
- y is assumed to be a MATLAB workspace vector.
- z is assumed to be a MATLAB workspace scalar variable.

x, y, and z must be defined in the MATLAB workspace prior to starting the simulation; otherwise a runtime error is generated at the execution time of the transition.

A MATLAB function named my_func is called with these arguments:

1 x(1,3)

2 y(3)

3 z

4 d1

5 d2

6 string 'abcdefgh'

The result of my_func() (if it is a scalar) is assigned to element (5, 6, 7) of a multi-dimensional matrix v in the MATLAB workspace. If v does not exist prior to the execution of this statement, then it is automatically created by MATLAB workspace.

If my_func() returns a vector, the first element is assigned to v(5,6,7). If it is a structure, a cell array, or a string, the result is undefined.

Choosing Between the ml() function and the ml Name Space Operator

It is recommended to use the ml name space operator wherever possible. The ml name space operator is faster and more robust than the ml() function. If you need to work with MATLAB matrices instead of scalars, then use the ml() function.

In this example, the `ml()` function must be used to specify an array argument:

```
a = ml('my_function([1:4],%g)',d1);
```

`x` is a MATLAB workspace matrix. `my_function` is a MATLAB function that expects a vector as its first argument and a scalar as a second argument.

Data and Event Arguments

Unqualified data and event objects are assumed to be defined at the same level in the hierarchy as the reference to them in the action language. Stateflow will attempt to resolve the object name by searching up the hierarchy. If the data or event object is parented elsewhere in the hierarchy, you need to define the hierarchy path explicitly.

Arrays

Arrays specified as arguments to user written functions are essentially ignored by Stateflow and represented "as is" in the generated code.

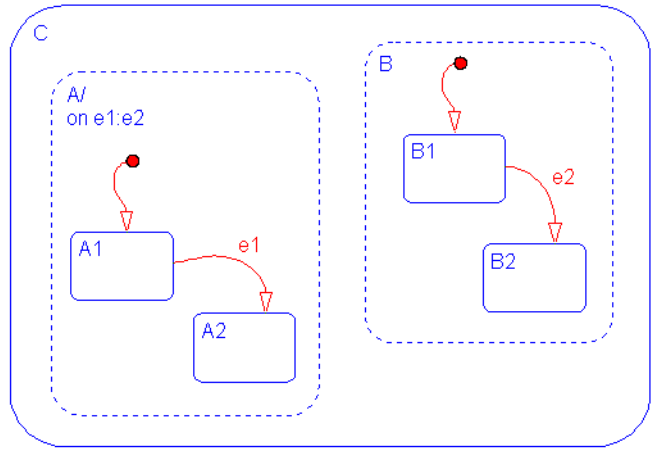
Event Broadcasting

You can specify an event to be broadcast in the action language. Events have hierarchy (a parent) and scope. The parent and scope together define a range of access to events. It is primarily the event's parent that determines who can trigger on the event (has receive rights). See "Table of Event Parent and Scope Combinations" on page 3-15 for more information.

Broadcasting an event in the action language is most useful as a means of synchronization amongst AND (parallel) states. Recursive event broadcasts can lead to definition of cyclic behavior. Cyclic behavior can be detected only during simulation.

Example: Event Broadcast State Action

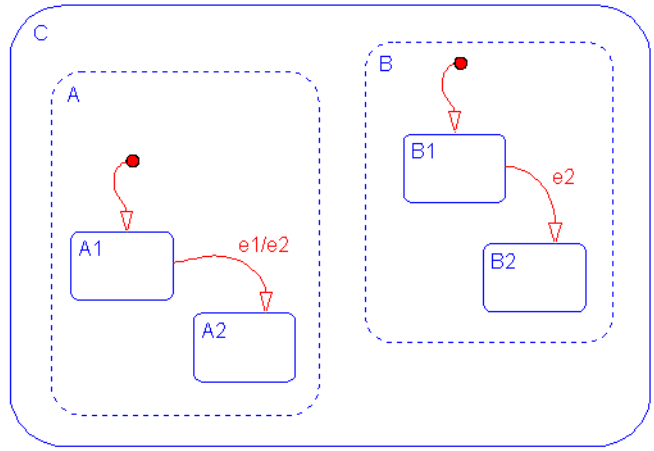
This is an example of the event broadcast state action notation.



See the section titled “Example: Event Broadcast State Action” on page 7-45 for information on the semantics of this notation.

Example: Event Broadcast Transition Action

This is an example of the event broadcast transition action notation.



See the section titled “Example: Event Broadcast Transition Action (Nested Event Broadcast)” on page 7-50 for information on the semantics of this notation.

Directed Event Broadcasting

You can specify a directed event broadcast in the action language. Using a directed event broadcast, you can broadcast a specific event to a specific receiver state. Directed event broadcasting is a more efficient means of synchronization amongst AND (parallel) states. Using directed event broadcasting improves the efficiency of the generated code. As is true in event broadcasting, recursive event broadcasts can lead to definition of cyclic behavior.

The format of the directed broadcast is:

```
send(event_name,state_name)
```

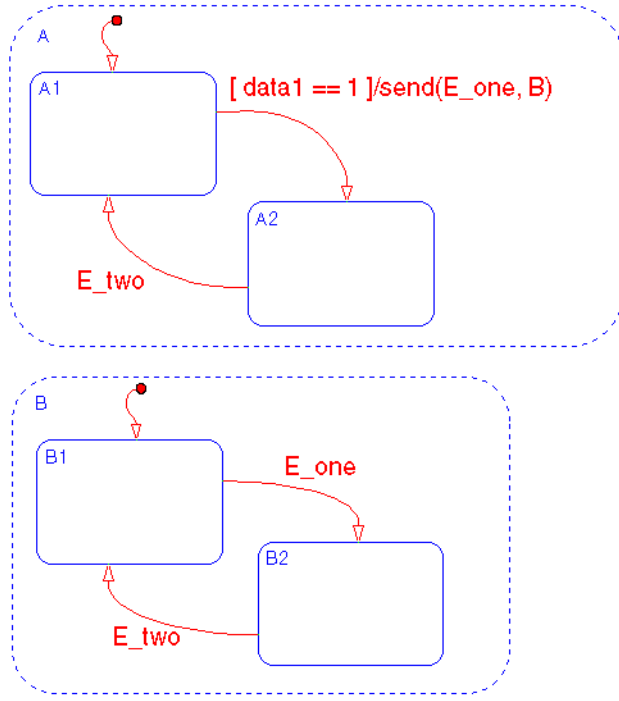
where `event_name` is broadcast to `state_name` (and any offspring of that state in the hierarchy). The `state_name` argument can include a full hierarchy path. For example,

```
send(event_name, chart_name.state_name1.state_name2)
```

The `state_name` specified must be active at the time the `send` is executed for the `state_name` to receive and potentially act on the directed event broadcast.

Example: Directed Event Broadcast Using send

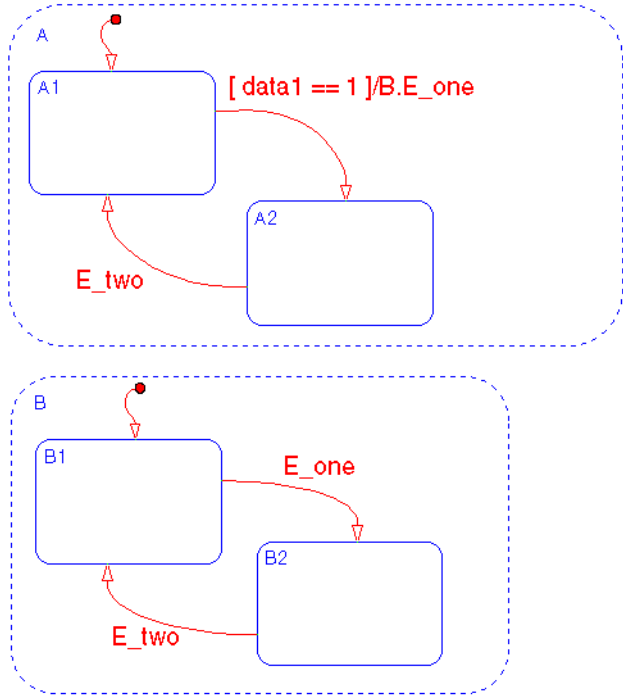
This is an example of a directed event broadcast using the `send(event_name,state_name)` transition action as a transition action:



See “Example: Directed Event Broadcasting Using Qualified Event Names” on page 7-60 for information on the semantics of this notation.

Example: Directed Event Broadcast Using Qualified Event Names

This example shows an alternate way to specify the same directed event broadcast as the previous example using a qualified event name.



See “Example: Directed Event Broadcasting Using Qualified Event Names” on page 7-60 for information on the semantics of this notation.

Conditions

You sometimes want transitions or actions associated with transitions to take place only if a certain condition is true. Conditions are placed within [ ]. These are some guidelines for defining conditions:

- The expression must be a Boolean expression of some kind. The condition must evaluate to either true (1) or false(0).
- The expression can consist of:
  - Boolean operators, which make comparisons between data and numeric values
  - Any function that returns a Boolean value
  - The `In(`*`state_name`*`)` condition function that is evaluated as true when the state specified as the argument is active. The full state name, including any ancestor states, must be specified to avoid ambiguity.
- The condition expression should not call a function that causes the Stateflow diagram to change state or modify any variables.
- Boolean expressions can be grouped using & for expressions with AND relationships and | for expressions with OR relationships.
- Assignment statements are not valid condition expressions.
- Unary increment and decrement actions are not valid condition expressions.

Time Symbol

You can use the letter `t` to represent absolute time in simulation targets. This simulation time is inherited from Simulink.

For example, the condition `[t - On_time > Duration]` specifies that the condition is true if the value of `On_time` subtracted from the simulation time `t`, is greater than the value of `Duration`.

The meaning of `t` for nonsimulation targets is undefined since it is dependent upon the specific application and target hardware.

Literals

Place action language you want the parser to ignore but you want to appear as entered in the generated code within $ characters. For example:

```
$
ptr -> field = 1.0;
$
```

The parser is completely disabled during the processing of anything between the $ characters. Frequent use of literals is discouraged.

Continuation Symbols

Enter the characters ... at the end of a line to indicate the expression continues on the next line.

Comments

These comment formats are supported:

- `%` MATLAB comment line
- `//` C++ comment line
- `/* C comment line */`

Use of the Semicolon

Omitting the semicolon after an expression displays the results of the expression in the MATLAB command window. If you use a semicolon, the results are not displayed.

7

Semantics

Overview 7-2

Event-Driven Effects on Semantics 7-5

Transitions to and from Exclusive (OR) States 7-8

Condition Actions 7-13

Default Transitions 7-18

Inner Transitions 7-25

Connective Junctions 7-33

Event Actions 7-43

Parallel (AND) States 7-45

Directed Event Broadcasting 7-58

Execution Order 7-62

Semantic Rules Summary 7-67

Overview

What Are Semantics?

Semantics describe how the notation is interpreted and implemented. A completed Stateflow diagram communicates how the system will behave. A Stateflow diagram contains actions associated with transitions and states. The semantics describe in what sequence these actions take place during Stateflow diagram execution.

Knowledge of the semantics is important to make sound Stateflow diagram design decisions for code generation. Different use of notations results in different ordering of simulation and generated code execution.

Stateflow semantics consist of rules for:

- Event broadcasting
- Processing states
- Processing transitions
- Taking transition paths

The details of Stateflow semantics are described largely by examples in this chapter. The examples cover a range of various notations and combinations of state and transition actions.

See “Semantic Rules Summary” on page 7-67 for a summary of the semantics.

List of Semantic Examples

This is a list of the semantic examples provided in this chapter.

Contents of section “Transitions to and from Exclusive (OR) States” on page 7-8

- “Example: Processing of One Event” on page 7-8
- “Example: Processing of a Second Event” on page 7-9
- “Example: Processing of a Third Event” on page 7-10
- “Example: Transition from a Substate to a Substate” on page 7-11

Contents of section “Condition Actions” on page 7-13

- “Example: Actions Specified as Condition Actions” on page 7-13
- “Example: Actions Specified as Condition and Transition Actions” on page 7-14
- “Example: Using Condition Actions in For Loop Construct” on page 7-15
- “Example: Using Condition Actions to Broadcast Events to Parallel (AND) States” on page 7-16
- “Example: Cyclic Behavior to Avoid When Using Condition Actions” on page 7-17

Contents of section “Default Transitions” on page 7-18

- “Example: Default Transition in an Exclusive (OR) Decomposition” on page 7-18
- “Example: Default Transition to a Junction” on page 7-20
- “Example: Default Transition and a History Junction” on page 7-21
- “Example: Labeled Default Transitions” on page 7-23

Contents of section “Inner Transitions” on page 7-25

- “Example: Processing One Event Within an Exclusive (OR) State” on page 7-25
- “Example: Processing a Second Event Within an Exclusive (OR) State” on page 7-26
- “Example: Processing a Third Event Within an Exclusive (OR) State” on page 7-27
- “Example: Processing One Event with an Inner Transition to a Connective Junction” on page 7-28
- “Example: Processing a Second Event with an Inner Transition to a Connective Junction” on page 7-29
- “Example: Inner Transition to a History Junction” on page 7-31

Contents of section “Connective Junctions” on page 7-33

- “Example: If-Then-Else Decision Construct” on page 7-33
- “Example: Self Loop” on page 7-34
- “Example: For Loop Construct” on page 7-35
- “Example: Flow Diagram Notation” on page 7-37
- “Example: Transitions from a Common Source to Multiple Destinations” on page 7-39
- “Example: Transitions from Multiple Sources to a Common Destination” on page 7-40
- “Example: Transitions from a Source to a Destination Based on a Common Event” on page 7-41

Contents of section “Event Actions” on page 7-43

- “Example: Event Actions and Superstates” on page 7-43

Contents of section “Parallel (AND) States” on page 7-45

- “Example: Event Broadcast State Action” on page 7-45
- “Example: Event Broadcast Transition Action (Nested Event Broadcast)” on page 7-50
- “Example: Event Broadcast Condition Action” on page 7-54

Contents of section “Directed Event Broadcasting” on page 7-58

- “Example: Directed Event Broadcast Using send” on page 7-58
- “Example: Directed Event Broadcasting Using Qualified Event Names” on page 7-60

Event-Driven Effects on Semantics

What Does Event-Driven Mean?

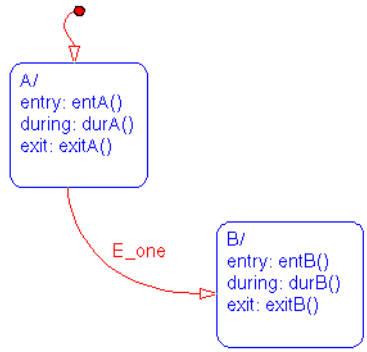
The Stateflow diagram executes only when an event occurs; an event occurs and the Stateflow diagram is awakened to respond to the event. Exactly what executes depends on the circumstances when the event occurs. Actions that are to take place based on an event are atomic to that event. Once an action is initiated, it is completed.

Top-Down Processing of Events

When an event occurs, it is processed from the top or root of the Stateflow diagram down through the hierarchy of the Stateflow diagram. At each level in the hierarchy, any `during` and `on` *`event_name`* actions for the active state are executed and completed and then a check for the existence of a valid explicit or implicit transition among the children of the state is conducted. The examples in this chapter demonstrate the top-down processing of events.

Semantics of Active and Inactive States

This example shows the semantics of active and inactive states.

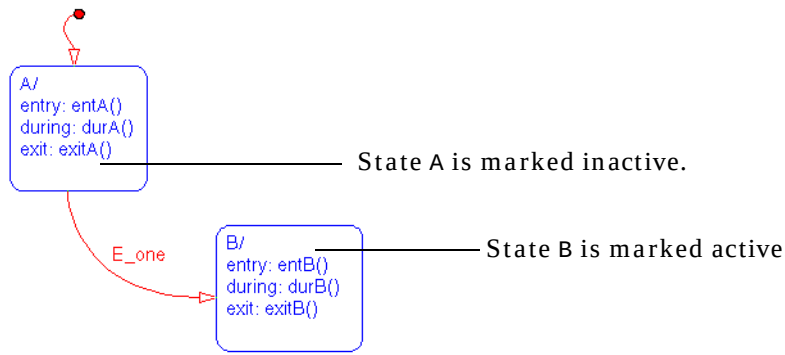


Initially the Stateflow diagram is asleep and both states are inactive. An event occurs and the Stateflow diagram is awakened. This is the semantic sequence.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of the event. A valid default transition to state `A` is detected.
2. State `A` is marked active.
3. State `A` `entry` actions execute and complete (`entA()`).
4. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

Event `E_one` occurs and the Stateflow diagram is awakened. State `A` is active. This is the semantic sequence.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A valid transition is detected from state `A` to state `B`.
2. State `A` `exit` actions execute and complete (`exitA()`).
3. State `A` is marked inactive.
4. State `B` is marked active.
5. State `B` `entry` actions execute and complete (`entB()`).
6. The Stateflow diagram goes back to sleep, to be awakened by the next event.



Semantics of State Actions

An `entry` action is executed as a result of any transition into the state. The state is marked active before its `entry` action is executed and completed.

A `during` action executes to completion when that state is active and an event occurs that does not result in an exit from that state. An `on` *event_name* action executes to completion when the event specified, *event_name*, occurs and that state is active. An active state executes its `during` and `on` *event_name* actions before processing any of its children's valid transitions. `During` and `on` *event_name* actions are processed based on their order of appearance in the state label.

An `exit` action is executed as a result of any transition out of the state. The state is marked inactive after the `exit` action has executed and completed.

Semantics of Transitions

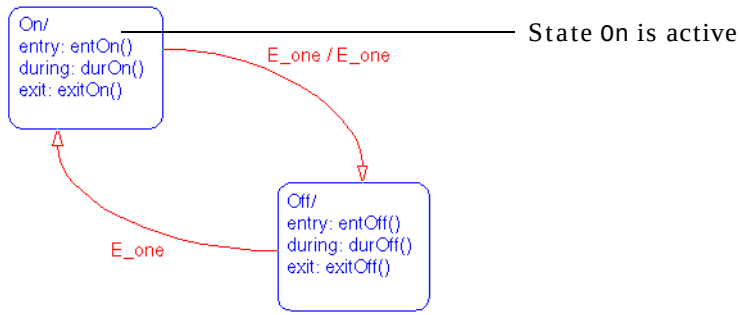
Transitions play a large role in defining the animation or execution of a system. Transitions have sources and destinations; thus any actions associated with the sources or destinations are related to the transition that joins them. The type of the source and destination is equally important to define the semantics.

The examples provided in this chapter show how the semantics are defined.

Transitions to and from Exclusive (OR) States

Example: Processing of One Event

This example shows the semantics of a simple transition focusing on the implications of states being active or inactive.



Initially the Stateflow diagram is asleep. State `On` and state `Off` are OR states. State `On` is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A valid transition from state `On` to state `Off` is detected.
2. State `On` `exit` actions execute and complete (`ExitOn()`).
3. State `On` is marked inactive.
4. The event `E_one` is broadcast as the transition action. The second generation of event `E_one` is processed but because neither state is active, it has no effect. (Had a valid transition been possible as a result of the broadcast of

E_one, the processing of the first broadcast of E_one would be preempted by the second broadcast of E_one.)

5 State Off is marked active.

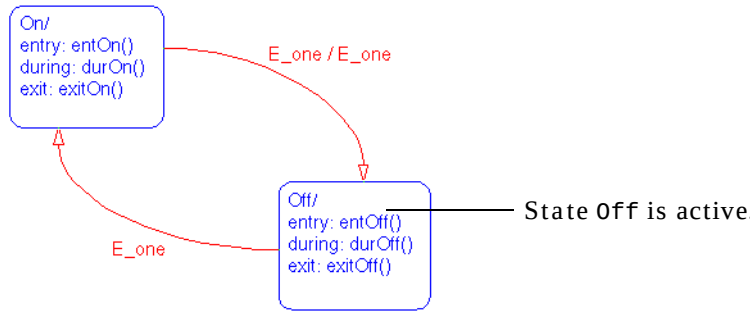
6 State Off entry actions execute and complete (entOff()).

7 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with event E_one when state On was active.

Example: Processing of a Second Event

Using the same example, what happens when the next event, E_one, occurs?



Again, initially the Stateflow diagram is asleep. State Off is active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed

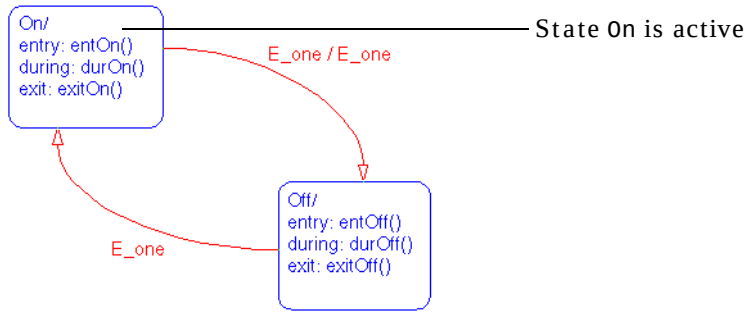
from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A valid transition from state `Off` to state `On` is detected.
2. State `Off` `exit` actions execute and complete (`exitOff()`).
3. State `Off` is marked inactive.
4. State `On` is marked active.
5. State `On` `entry` actions execute and complete (`entOn()`).
6. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with the second event `E_one` when state `Off` was active.

Example: Processing of a Third Event

Using the same example, what happens when a third event, `E_two`, occurs?



Again, initially the Stateflow diagram is asleep. State `On` is active. Event `E_two` occurs and awakens the Stateflow diagram. Event `E_two` is processed from the

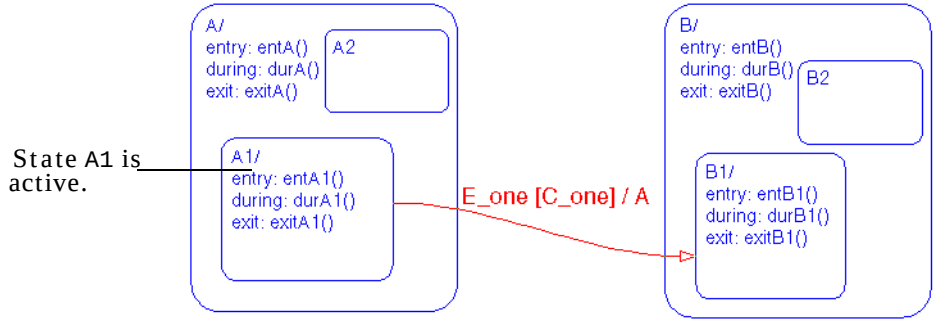
root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_two`. There is none.
2. State `On` `during` actions execute and complete (`durOn()`).
3. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with event `E_two` when State `On` was active.

Example: Transition from a Substate to a Substate

This example shows the semantics of a transition from an OR substate to an OR substate.



Initially the Stateflow diagram is asleep. State `A.A1` is active. Event `E_one` occurs and awakens the Stateflow diagram. Condition `C_one` is true. Event

E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

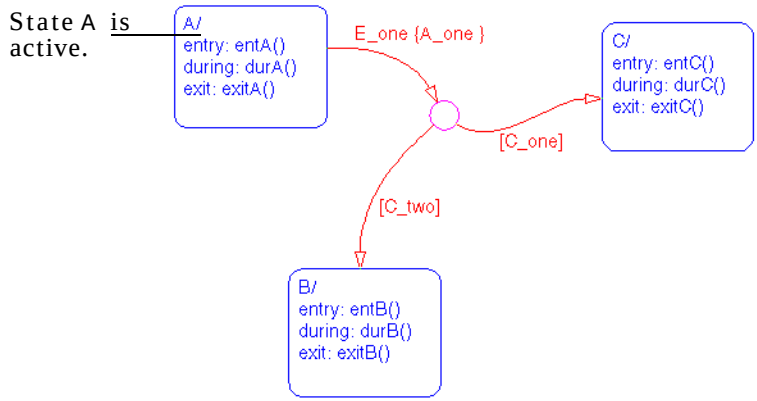
1. The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid implicit transition from state A.A1 to state B.B1. (Condition C_one is true.)
2. State A executes and completes during actions (durA( )).
3. State A.A1 executes and completes exit actions (exitA1( )).
4. State A.A1 is marked inactive.
5. State A executes and completes exit actions (exitA( )).
6. State A is marked inactive.
7. The transition action, A, is executed and completed.
8. State B is marked active.
9. State B executes and completes entry actions (entB( )).
10. State B.B1 is marked active.
11. State B.B1 executes and completes entry actions (entB1( )).
12. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Condition Actions

Example: Actions Specified as Condition Actions

This example shows the semantics of a simple condition action in a multiple segment transition.



Initially the Stateflow diagram is asleep. State `A` is active. Event `E_one` occurs and awakens the Stateflow diagram. Conditions `C_one` and `C_two` are false. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1 The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A valid transition segment from state `A` to a connective junction is detected. The condition action, `A_one`, is detected on the valid

transition segment and is immediately executed and completed. State A is still active.

2 Since the conditions on the transition segments to possible destinations are false, none of the complete transitions is valid.

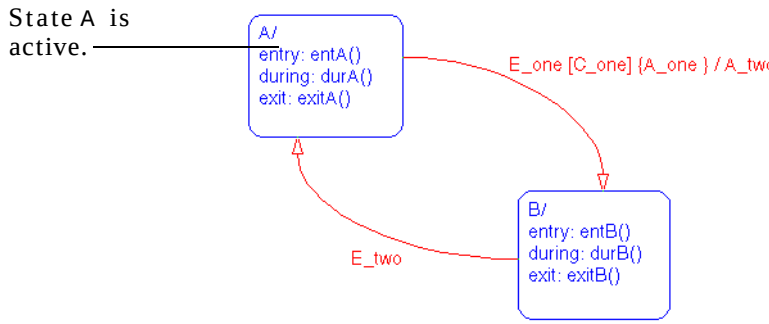
3 State A remains active. State A during action executes and completes (durA( )).

4 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with event E_one when state A was active.

Example: Actions Specified as Condition and Transition Actions

This example shows the semantics of a simple condition and transition action specified on a transition from one exclusive (OR) state to another.



Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs and awakens the Stateflow diagram. Condition C_one is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. A valid transition from state A to state B is detected. The

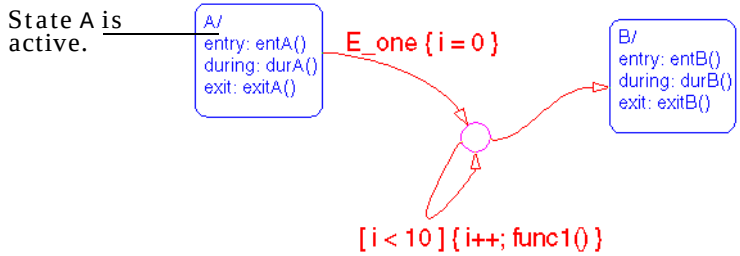
condition, C_one is true. The condition action, A_one, is detected on the valid transition and is immediately executed and completed. State A is still active.

2. State A exit actions execute and complete (ExitA( )).
3. State A is marked inactive.
4. The transition action, A_two, is executed and completed.
5. State B is marked active.
6. State B entry actions execute and complete (entB( )).
7. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of the Stateflow diagram associated with event E_one when state A was active.

Example: Using Condition Actions in For Loop Construct

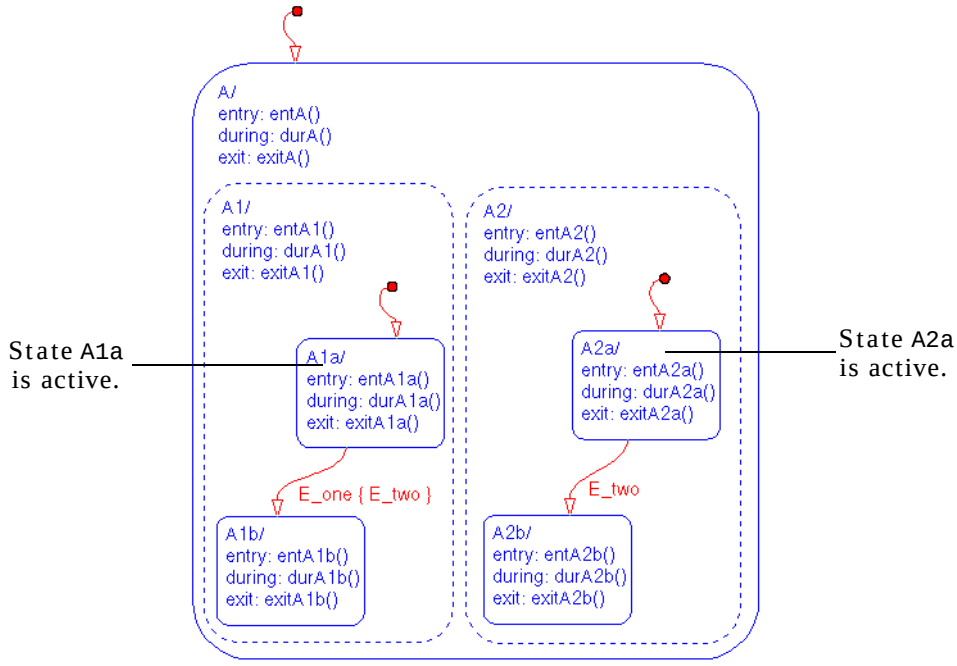
Condition actions and connective junctions are used to design a for loop construct. This example shows the use of a condition action and connective junction to create a for loop construct.



See “Example: For Loop Construct” on page 7-35 to see the semantics of this example.

Example: Using Condition Actions to Broadcast Events to Parallel (AND) States

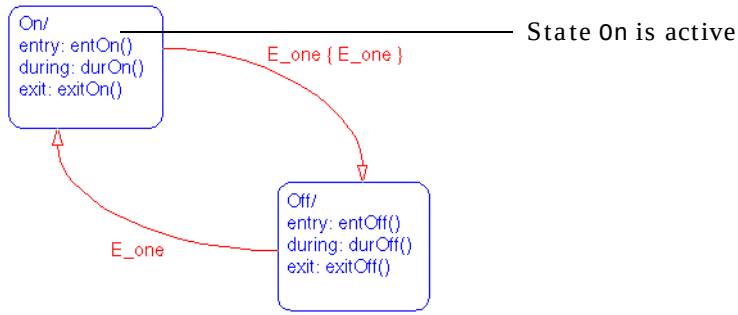
Condition actions can be used to broadcast events immediately to parallel (AND) states. This example shows this use.



See “Example: Event Broadcast Condition Action” on page 7-54 to see the semantics of this example.

Example: Cyclic Behavior to Avoid When Using Condition Actions

This example shows a notation to avoid when using event broadcasts as condition actions because the semantics result in cyclic behavior.



Initially the Stateflow diagram is asleep. State `On` is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A valid transition from state `On` to state `Off` is detected. A condition action, broadcast of event `E_one`, is detected on the valid transition and is immediately executed. State `On` is still active.

   a. The broadcast of event `E_one` awakens the Stateflow diagram a second time. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. The transition from state `On` to state `Off` is still valid. The condition action, broadcast of event `E_one`, is immediately executed again.

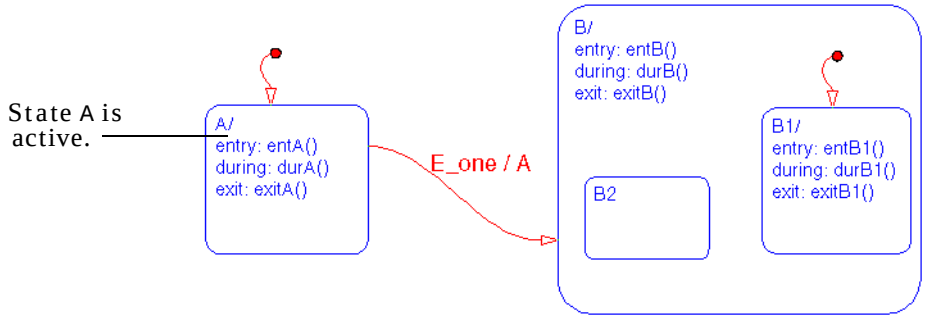
2. Step 1a continues to execute in a cyclical manner. The transition label indicating a trigger on the same event as the condition action broadcast event results in unrecoverable cyclic behavior.

This sequence never completes when event `E_one` is broadcast and state `On` is active.

Default Transitions

Example: Default Transition in an Exclusive (OR) Decomposition

This example shows a transition from an OR state to a superstate with exclusive (OR) decomposition, where a default transition to a substate is defined.



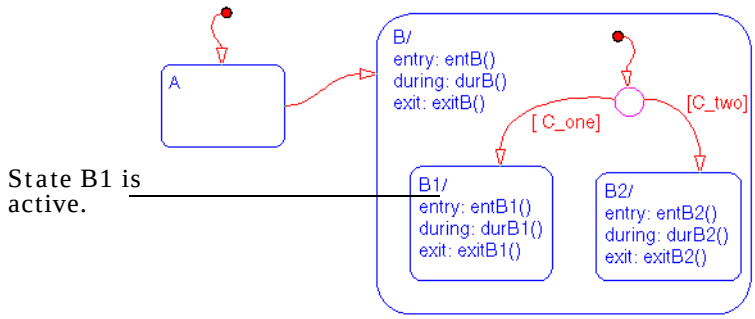
Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition from state A to superstate B.
2. State A `exit` actions execute and complete (`exitA()`).
3. State A is marked inactive.
4. The transition action, A, is executed and completed.
5. State B is marked active.
6. State B `entry` actions execute and complete (`entB()`).
7. State B detects a valid default transition to state B.B1.
8. State B.B1 is marked active.
9. State B.B1 `entry` actions execute and complete (`entB1()`).
10. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Example: Default Transition to a Junction

This example shows the semantics of a default transition to a connective junction.



Initially the Stateflow diagram is asleep. State `B.B1` is active. An event occurs and awakens the Stateflow diagram. Condition `[C_two]` is true. The event is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of any event. There is none.

2. State `B` `during` actions execute and complete (`durB()`).

3. State `B` checks for valid transitions and detects a valid transition segment from a default transition to a connective junction. The transition segments beginning from a twelve o'clock position and progressing clockwise on the connective junction are evaluated for validity. The first transition segment

evaluated, labeled with [C_two] is valid. The complete transition from the default transition to state B.B2 is valid.

4 State B.B1 executes and completes exit actions (exitB1( )).

5 State B.B1 is marked inactive.

6 State B.B2 is marked active.

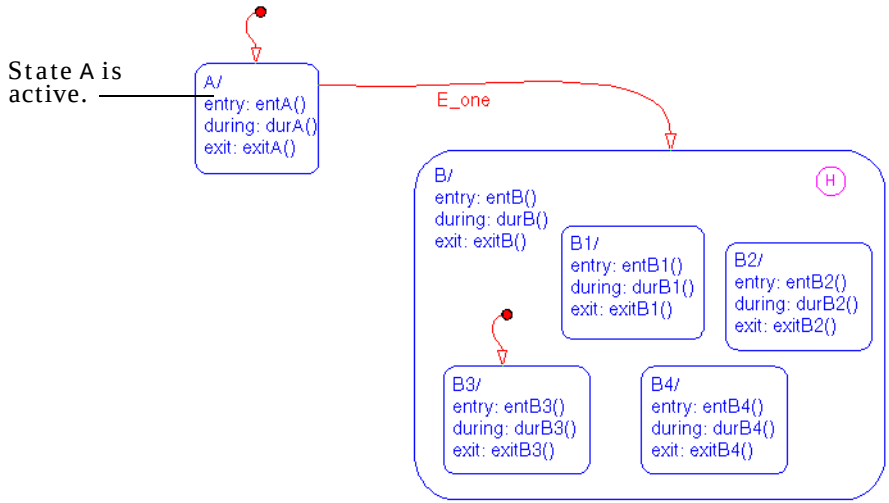
7 State B.B2 executes and completes entry actions (entB2( )).

8 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with the occurrence of any event.

Example: Default Transition and a History Junction

This example shows the semantics of a superstate and a history junction.



Initially the Stateflow diagram is asleep. State A is active. There is a history junction and state B4 was the last active substate of superstate B. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the

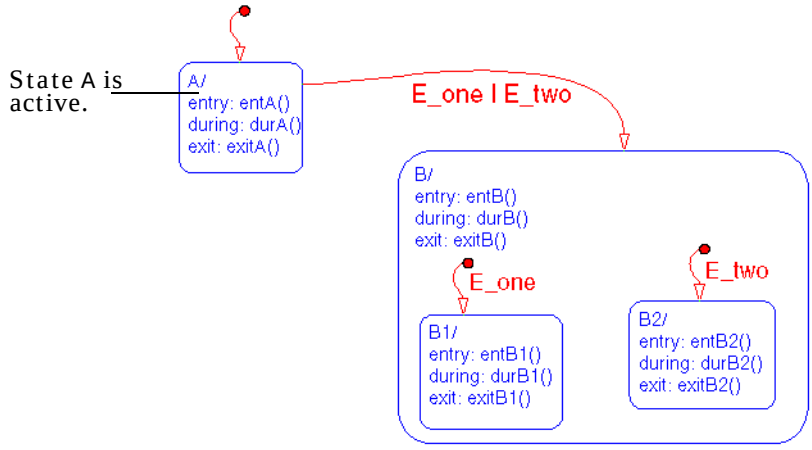
root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is valid transition from state `A` to superstate `B`.
2. State `A` `exit` actions execute and complete (`exitA()`).
3. State `A` is marked inactive.
4. State `B` is marked active.
5. State `B` `entry` actions execute and complete (`entB()`).
6. State `B` detects and uses the history junction to determine which substate is the destination of the transition into the superstate. The history junction indicates substate `B.B4` was the last active substate, and thus the destination of the transition.
7. State `B.B4` is marked active.
8. State `B.B4` `entry` actions execute and complete (`entB4()`).
9. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

Example: Labeled Default Transitions

This example shows the use of a default transition with a label.



Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs awakening the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

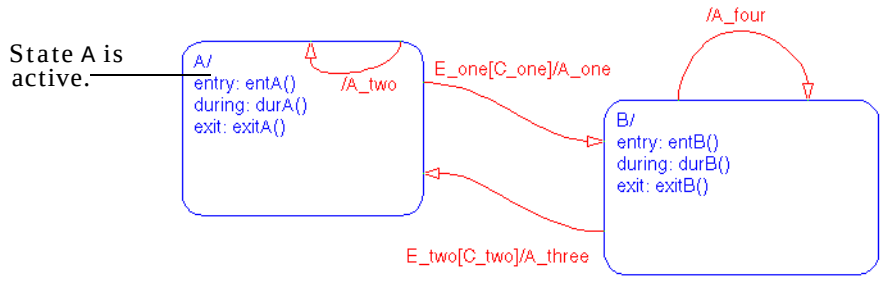
1. The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition from state A to superstate B. A pipe is used to represent that the transition is valid if event E_one or E_two occurs.
2. State A exit actions execute and complete (exitA()).
3. State A is marked inactive.
4. State B is marked active.
5. State B entry actions execute and complete (entB()).
6. State B detects a valid default transition to state B.B1. The default transition is valid as a result of E_one.
7. State B.B1 is marked active.
8. State B.B1 entry actions execute and complete (entB1()).
9. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Inner Transitions

Example: Processing One Event Within an Exclusive (OR) State

This example shows the semantics of an inner transition.



Initially the Stateflow diagram is asleep. State `A` is active. Event `E_one` occurs and awakens the Stateflow diagram. Condition `[C_one]` is false. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. A potentially valid transition from state `A` to state `B` is detected. However the transition is not valid because `[C_one]` is false.

2. State `A` `during` actions execute and complete (`durA()`).

3. State `A` checks its children for a valid transition and detects a valid inner transition.

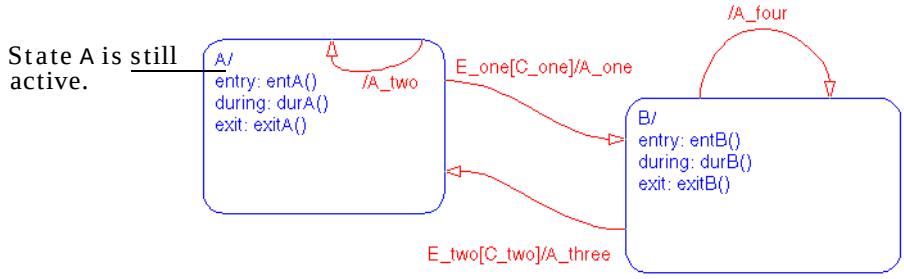
4. State `A` remains active. The inner transition action, `A_two`, is executed and completed. Because it is an inner transition, state `A`'s `exit` and `entry` actions are not executed.

5. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

Example: Processing a Second Event Within an Exclusive (OR) State

Using the same example, what happens when a second event, E_one, occurs?



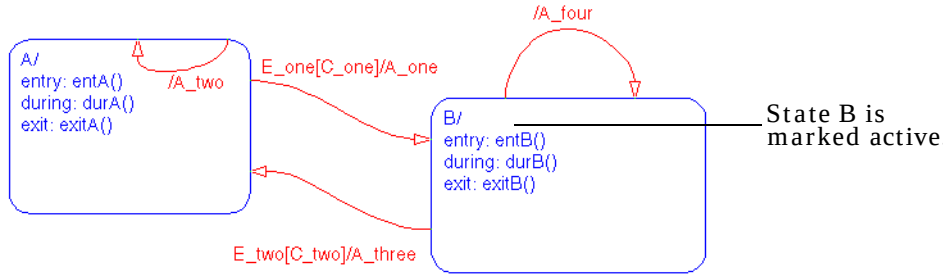
Initially the Stateflow diagram is asleep. State A is still active. Event E_one occurs and awakens the Stateflow diagram. Condition [C_one] is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. The transition from state A to state B is now valid because [C_one] is true.
2. State A exit actions execute and complete (exitA()).
3. State A is marked inactive.
4. The transition action A_one is executed and completed.
5. State B is marked active.
6. State B entry actions execute and complete (entB()).
7. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Example: Processing a Third Event Within an Exclusive (OR) State

Using the same example, what happens when a third event, `E_two`, occurs?



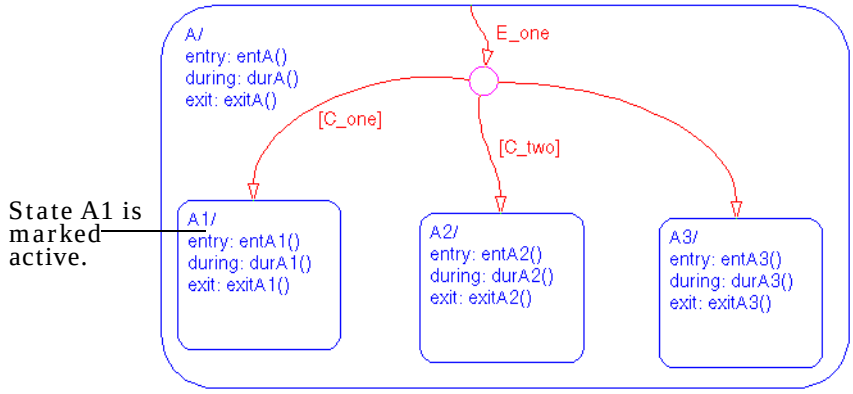
Initially the Stateflow diagram is asleep. State `B` is now active. Event `E_two` occurs and awakens the Stateflow diagram. Condition `[C_two]` is false. Event `E_two` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_two`. A potentially valid transition from state `B` to state `A` is detected. The transition is not valid because `[C_two]` is false. However, active state `B` has a valid self loop transition.
2. State `B` `exit` actions execute and complete (`exitB()`).
3. State `B` is marked inactive.
4. The self loop transition action, `A_four`, executes and completes.
5. State `B` is marked active.
6. State `B` `entry` actions execute and complete (`entB()`).
7. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_two`. This example shows the difference in semantics between inner transitions and self loop transitions.

Example: Processing One Event with an Inner Transition to a Connective Junction

This example shows the semantics of an inner transition to a connective junction.



Initially the Stateflow diagram is asleep. State `A1` is active. Event `E_one` occurs and awakens the Stateflow diagram. Condition `[C_two]` is true. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition at the root level, as a result of `E_one`. There is no valid transition.

2. State `A` `during` actions execute and complete (`durA()`).

3. State `A` checks itself for valid transitions and detects there is a valid inner transition to a connective junction. The conditions are evaluated to determine if one of the transitions is valid. The segments labeled with a condition are evaluated before the unlabeled segment. The evaluation starts from a twelve o'clock position on the junction and progresses in a clockwise manner.

Since [C_two] is true, the inner transition to the junction and then to state A.A2 is valid.

4 State A.A1 exit actions execute and complete (exitA1()).

5 State A.A1 is marked inactive.

6 State A.A2 is marked active.

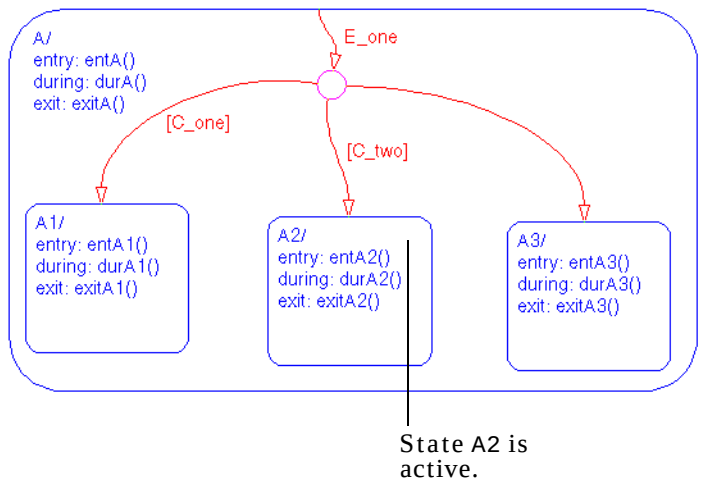
7 State A.A2 entry actions execute and complete (entA2()).

8 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one when condition C_two is true.

Example: Processing a Second Event with an Inner Transition to a Connective Junction

This example shows the semantics of an inner transition to a junction when a second event, E_one, occurs.



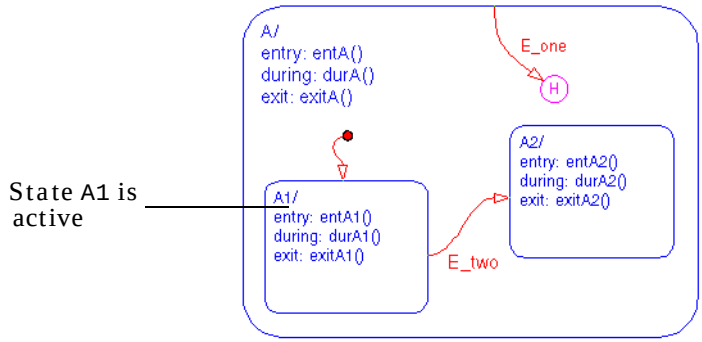
Initially the Stateflow diagram is asleep. State A2 is active. Event E_one occurs and awakens the Stateflow diagram. Neither [C_one] nor [C_two] is true. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition at the root level, as a result of E_one. There is no valid transition.
2. State A during actions execute and complete (durA()).
3. State A checks itself for valid transitions and detects a valid inner transition to a connective junction. The segments labeled with a condition are evaluated before the unlabeled segment. The evaluation starts from a twelve o'clock position on the junction and progresses in a clockwise manner. Since neither [C_one] nor [C_two] is true, the unlabeled transition segment is evaluated and is determined to be valid. The full transition from the inner transition to state A.A3 is valid.
4. State A.A2 exit actions execute and complete (exitA2()).
5. State A.A2 is marked inactive.
6. State A.A3 is marked active.
7. State A.A3 entry actions execute and complete (entA3()).
8. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one when neither [C_one] nor [C_two] is true.

Example: Inner Transition to a History Junction

This example shows the semantics of an inner transition to a history junction.



Initially the Stateflow diagram is asleep. State `A.A1` is active. There is history information since superstate `A` is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

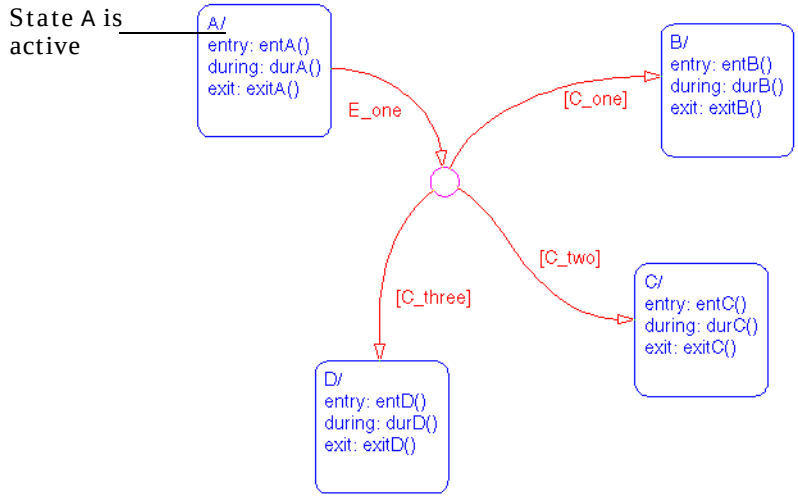
1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is no valid transition.
2. State `A` `during` actions execute and complete (`durA()`).
3. State `A` checks itself for valid transitions and detects there is a valid inner transition to a history junction. According to the semantics of history junctions, the last active state, `A.A1`, is the destination state.
4. State `A.A1` `exit` actions execute and complete (`exitA1()`).
5. State `A.A1` is marked inactive.
6. State `A.A1` is marked active.
7. State `A.A1` `entry` actions execute and complete (`entA1()`).
8. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one` when there is an inner transition to a history junction and state `A.A1` is active.

Connective Junctions

Example: If-Then-Else Decision Construct

This example shows the semantics of an `if-then-else` decision construct.



Initially the Stateflow diagram is asleep. State `A` is active. Event `E_one` occurs and awakens the Stateflow diagram. Condition [`C_two`] is true. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition segment from state `A` to the connective junction. The transition segments beginning from a twelve o'clock position on the connective junction are evaluated for validity. The first transition segment labeled with condition [`C_one`] is not valid. The next

transition segment labeled with the condition [C_two] is valid. The complete transition from state A to state C is valid.

2 State A executes and completes exit actions (exitA( )).

3 State A is marked inactive.

4 State C is marked active.

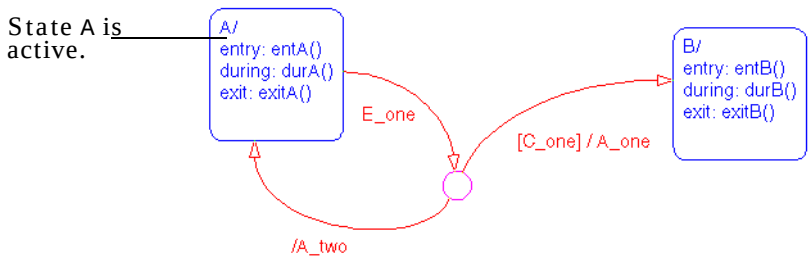
5 State C executes and completes entry actions (entC( )).

6 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Example: Self Loop

This example shows the semantics of a self loop using a connective junction.



Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs and awakens the Stateflow diagram. Condition [C_one] is false. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition segment from state A to the connective junction. The transition segment labeled with a condition and action is evaluated for validity. Since the condition [C_one] is not valid, the

complete transition from state A to state B is not valid. The transition segment from the connective junction back to state A is valid.

2 State A executes and completes exit actions (exitA()).

3 State A is marked inactive.

4 The transition action A_two is executed and completed.

5 State A is marked active.

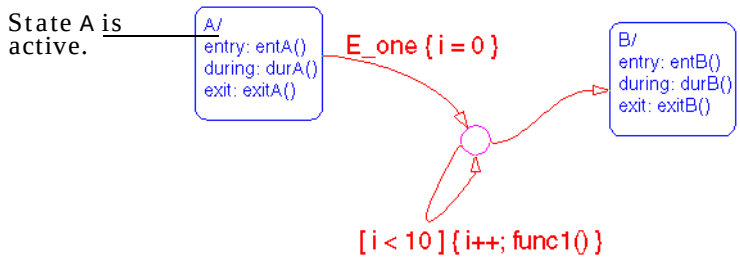
6 State A executes and completes entry actions (entA()).

7 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Example: For Loop Construct

This example shows the semantics of a for loop.



Initially the Stateflow diagram is asleep. State A is active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1 The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition segment from state A to the connective junction. The transition segment condition action, i = 0, is

executed and completed. Of the two transition segments leaving the connective junction, the transition segment that is a self loop back to the connective junction is evaluated next for validity. That segment takes priority in evaluation because it has a condition specified whereas the other segment is unlabeled.

2. The condition [`i < 10`] is evaluated as true. The condition actions, `i++`, and a call to `func1` are executed and completed until the condition becomes false. A connective junction is not a final destination; thus the transition destination remains to be determined.

3. The unconditional segment to state `B` is now valid. The complete transition from state `A` to state `B` is valid.

4. State `A` executes and completes `exit` actions (`exitA()`).

5. State `A` is marked inactive.

6. State `B` is marked active.

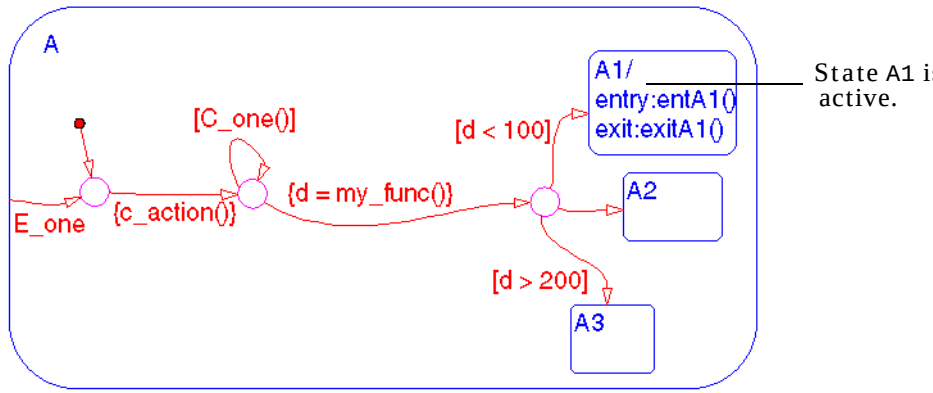
7. State `B` executes and completes `entry` actions (`entB()`).

8. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

Example: Flow Diagram Notation

This example shows the semantics of a Stateflow diagram that uses flow notation.



Initially the Stateflow diagram is asleep. State `A.A1` is active. The condition [C_one()] is initially true. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is no valid transition.

2. State `A` checks itself for valid transitions and detects a valid inner transition to a connective junction.

3. The next possible segments of the transition are evaluated. There is only one outgoing transition and it has a condition action defined. The condition action is executed and completed.

4. The next possible segments are evaluated. There are two outgoing transitions; one is a conditional self loop and the other is an unconditional transition segment. The conditional transition segment takes precedence. The condition `[C_one( )]` is tested and is true; the self loop is taken. Since a final

transition destination has not been reached, this self loop continues until `[C_one()]` is false. Assume that after five loops `[C_one()]` is false.

5. The next possible transition segment (to the next connective junction) is evaluated. It is an unconditional transition segment with a condition action. The transition segment is taken and the condition action, `{d=my_func()}`, is executed and completed. The returned value of `d` is 84.

6. The next possible transition segment is evaluated. There are three possible outgoing transition segments to consider. Two are conditional; one is unconditional. The segment labeled with the condition `[d<100]` is evaluated first based on the geometry of the two outgoing conditional transition segments. Since the return value of `d` is 84, the condition `[d<100]` is true and this transition (to the destination state `A.A1`) is valid.

7. State `A.A1` exit actions execute and complete (`exitA1()`).

8. State `A.A1` is marked inactive.

9. State `A.A1` is marked active.

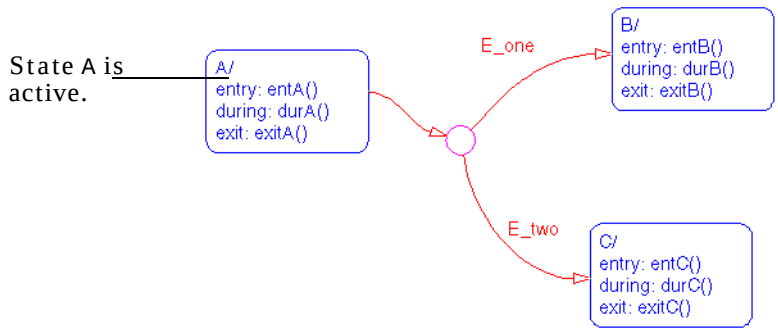
10. State `A.A1` `entry` actions execute and complete (`entA1()`).

11. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one`.

Example: Transitions from a Common Source to Multiple Destinations

This example shows the semantics of transitions from a common source to multiple destinations.



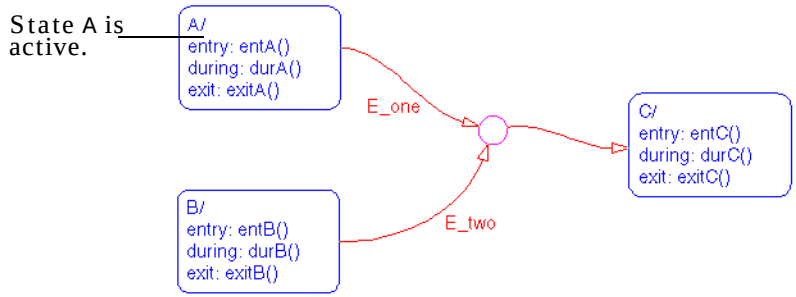
Initially the Stateflow diagram is asleep. State `A` is active. Event `E_two` occurs and awakens the Stateflow diagram. Event `E_two` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_two`. There is a valid transition segment from state `A` to the connective junction. Given that the transition segments are equivalently labeled, evaluation begins from a twelve o'clock position on the connective junction and progresses clockwise. The first transition segment labeled with event `E_one` is not valid. The next transition segment labeled with event `E_two` is valid. The complete transition from state `A` to state `C` is valid.
2. State `A` executes and completes `exit` actions (`exitA()`).
3. State `A` is marked inactive.
4. State `C` is marked active.
5. State `C` executes and completes `entry` actions (`entC()`).
6. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_two`.

Example: Transitions from Multiple Sources to a Common Destination

This example shows the semantics of transitions from multiple sources to a single destination.



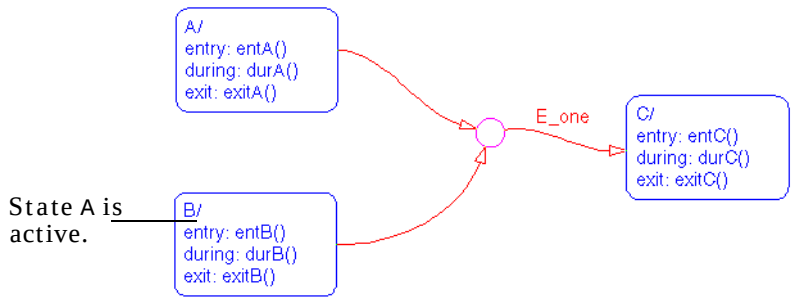
Initially the Stateflow diagram is asleep. State `A` is active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is a valid transition segment from state `A` to the connective junction and from the junction to state C.
2. State `A` executes and completes `exit` actions (`exitA()`).
3. State `A` is marked inactive.
4. State `C` is marked active.
5. State `C` executes and completes `entry` actions (`entC()`).
6. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Example: Transitions from a Source to a Destination Based on a Common Event

This example shows the semantics of transitions from multiple sources to a single destination based on the same event.



Initially the Stateflow diagram is asleep. State B is active. Event E_one occurs and awakens the Stateflow diagram. Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

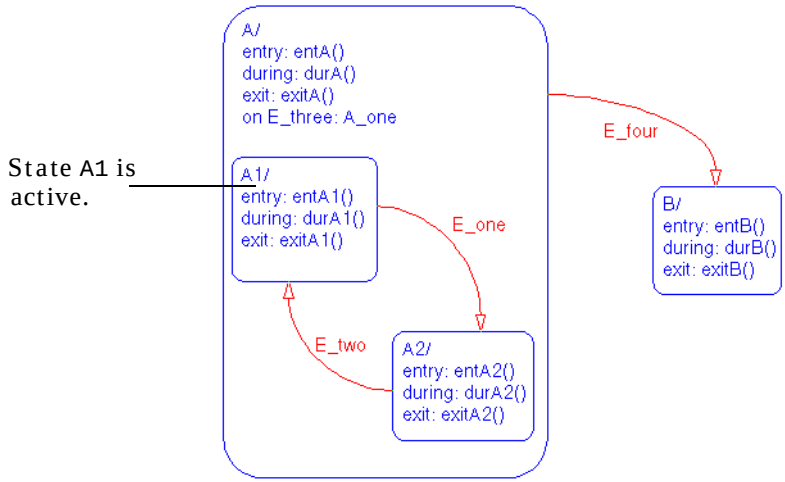
1. The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is a valid transition segment from state B to the connective junction and from the junction to state C.
2. State B executes and completes exit actions (exitB()).
3. State B is marked inactive.
4. State C is marked active.
5. State C executes and completes entry actions (entC()).
6. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event E_one.

Event Actions

Example: Event Actions and Superstates

This example shows the semantics of event actions within superstates.



Initially the Stateflow diagram is asleep. State `A.A1` is active. Event `E_three` occurs and awakens the Stateflow diagram. Event `E_three` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

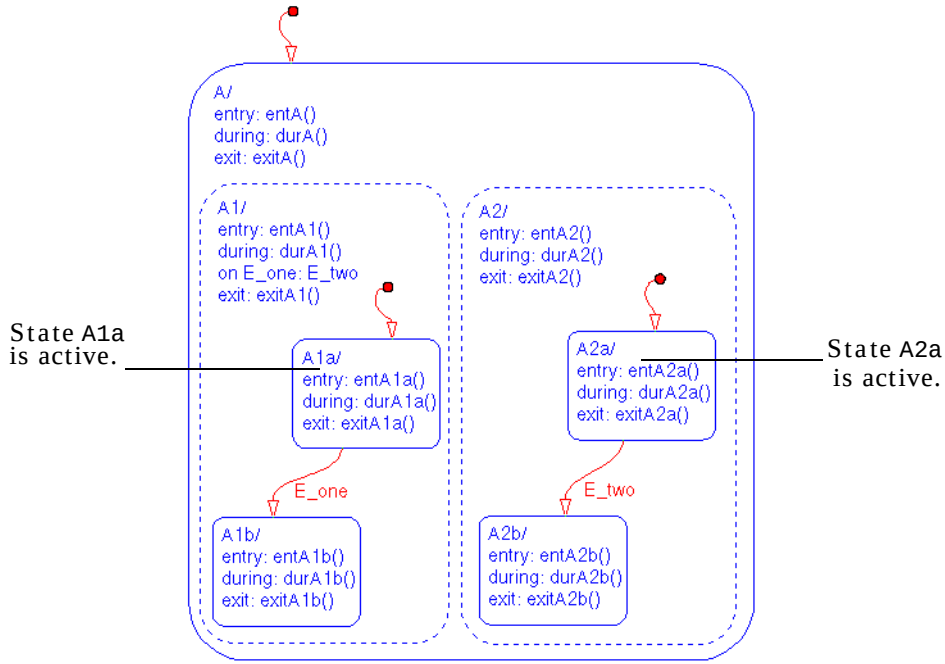
1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_three`. There is no valid transition.
2. State `A` executes and completes `during` actions (`durA()`).
3. State `A` executes and completes the `on` event `E_three` action (`A_one`).
4. State `A` checks its children for valid transitions. There are no valid transitions.
5. State `A1` executes and completes `during` actions (`durA1()`).
6. The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_three`.

Parallel (AND) States

Example: Event Broadcast State Action

This example shows the semantics of event broadcast state actions.



Initially the Stateflow diagram is asleep. Parallel substates A.A1.A1a and A.A2.A2a are active. Event E_one occurs and awakens the Stateflow diagram.

Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition at the root level, as a result of E_one. There is no valid transition.

2. State A executes and completes during actions (durA( )).

3. State A's children are parallel (AND) states. They are evaluated and executed from left to right and top to bottom. State A.A1 is evaluated first. State A.A1 executes and completes during actions (durA1( )). State A.A1 executes and completes the on E_one action and broadcasts event E_two. during and on *event_name* actions are processed based on their order of appearance in the state label.

   a. The broadcast of event E_two awakens the Stateflow diagram a second time. The Stateflow diagram root checks to see if there is a valid transition as a result of E_two. There is no valid transition.

   b. State A executes and completes during actions (durA()).

   c. State A checks its children for valid transitions. There are no valid transitions.

   d. State A' s children are evaluated starting with state A.A1. State A.A1 executes and completes during actions (durA1( )). State A.A1 is evaluated for

valid transitions. There are no valid transitions as a result of E_two within state A1.

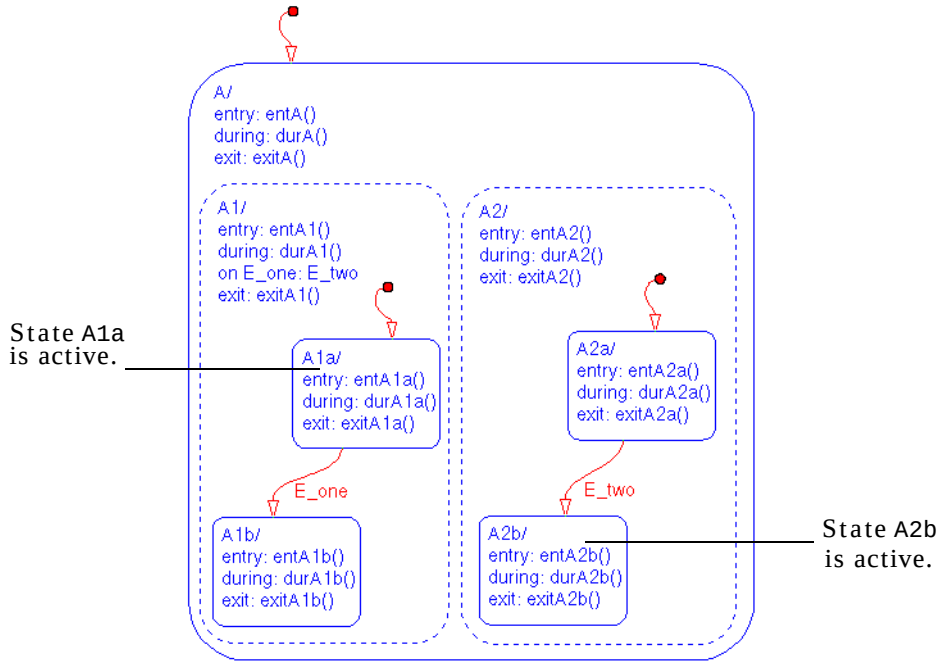
e State A.A2 is evaluated. State A.A2 executes and completes during actions (durA2()). State A.A2 checks for valid transitions. State A.A2 has a valid transition as a result of E_two from state A.A2.A2a to state A.A2.A2b.

f State A.A2.A2a exit actions execute and complete (exitA2a()).

g State A.A2.A2a is marked inactive.

h State A.A2.A2b is marked active.

i State A.A2.A2b entry actions execute and complete (entA2b()). The Stateflow diagram activity now looks like this.



4 The processing of E_one continues once the on event broadcast of E_two has been processed. State A.A1 checks for any valid transitions as a result of event E_one. There is a valid transition from state A.A1.A1a to state A.A1.A1b.

5 State A.A1.A1a executes and completes exit actions (exitA1a).

6 State A.A1.A1a is marked inactive.

7 State A.A1.A1b executes and completes entry actions (entA1b( )).

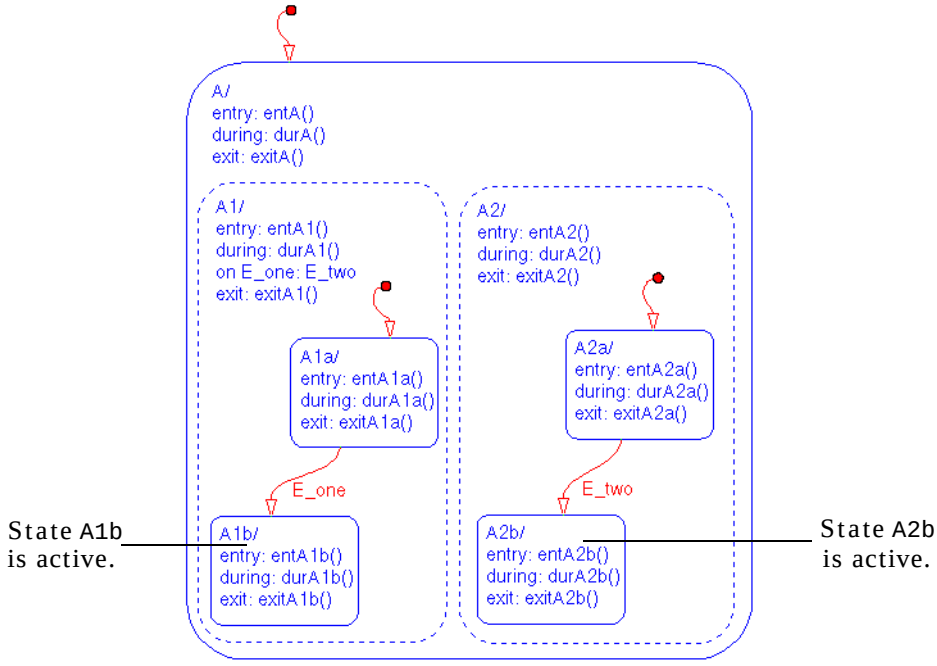
8 State A.A1.A1b is marked active.

9 Parallel state A.A2 is evaluated next. State A.A2 during actions execute and complete (durA2( )). There are no valid transitions as a result of E_one.

10 State A.A2.A2b, now active as a result of the processing of the on event broadcast of E_two, executes and completes during actions (durA2b( )).

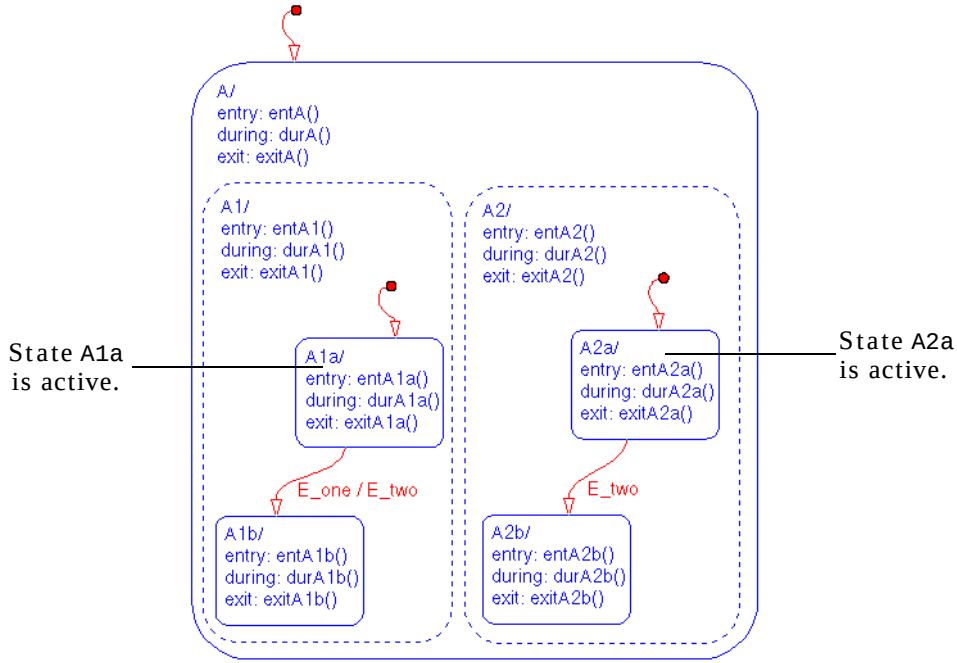
11 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one` and the on event broadcast to a parallel state of event `E_two`. The final Stateflow diagram activity looks like this.



Example: Event Broadcast Transition Action (Nested Event Broadcast)

This example shows the semantics of an event broadcast transition action that includes nested event broadcasts.



Initially the Stateflow diagram is asleep. Parallel substates A.A1.A1a and A.A2.A2a are active. Event E_one occurs and awakens the Stateflow diagram.

Start of event E_one processing

Event E_one is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of E_one. There is no valid transition.
2. State A executes and completes during actions (durA( )).
3. State A's children are parallel (AND) states. They are evaluated and executed from left to right and top to bottom. State A.A1 is evaluated first. State A.A1 executes and completes during actions (durA1( )).
4. State A.A1 checks for any valid transitions as a result of event E_one. There is a valid transition from state A.A1.A1a to state A.A1.A1b.
5. State A.A1.A1a executes and completes exit actions (exitA1a).

Start of event E_two processing (E_one processing is preempted)

6. State A.A1.A1a is marked inactive.
7. Transition action generating event E_two is executed and completed.
   - **a** The transition from state A1a to state A1b (as a result of event E_one) is now preempted by the broadcast of event E_two.
   - **b** The broadcast of event E_two awakens the Stateflow diagram a second time. The Stateflow diagram root checks to see if there is a valid transition as a result of E_two. There is no valid transition.
   - **c** State A executes and completes during actions (durA( )).
   - **d** State A' s children are evaluated starting with state A.A1. State A.A1 executes and completes during actions (durA1( )). State A.A1 is evaluated for

valid transitions. There are no valid transitions as a result of E_two within state A1.

e State A.A2 is evaluated. State A.A2 executes and completes during actions (durA2()). State A.A2 checks for valid transitions. State A.A2 has a valid transition as a result of E_two from state A.A2.A2a to state A.A2.A2b.

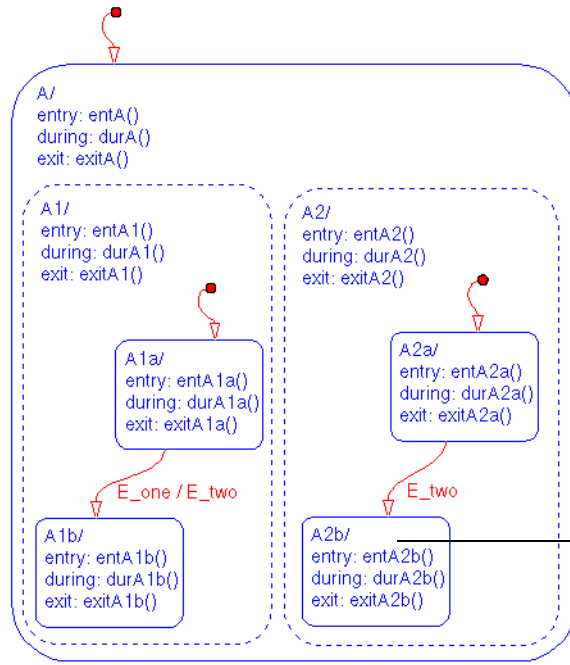
f State A.A2.A2a exit actions execute and complete (exitA2a()).

g State A.A2.A2a is marked inactive.

h State A.A2.A2b is marked active.

i State A.A2.A2b entry actions execute and complete (entA2b()). The Stateflow diagram activity now looks like this.

End of event E_two processing



State A2b is active.

Continuation of event E_one processing

8 State A.A1.A1b executes and completes `entry` actions (`entA1b()`).

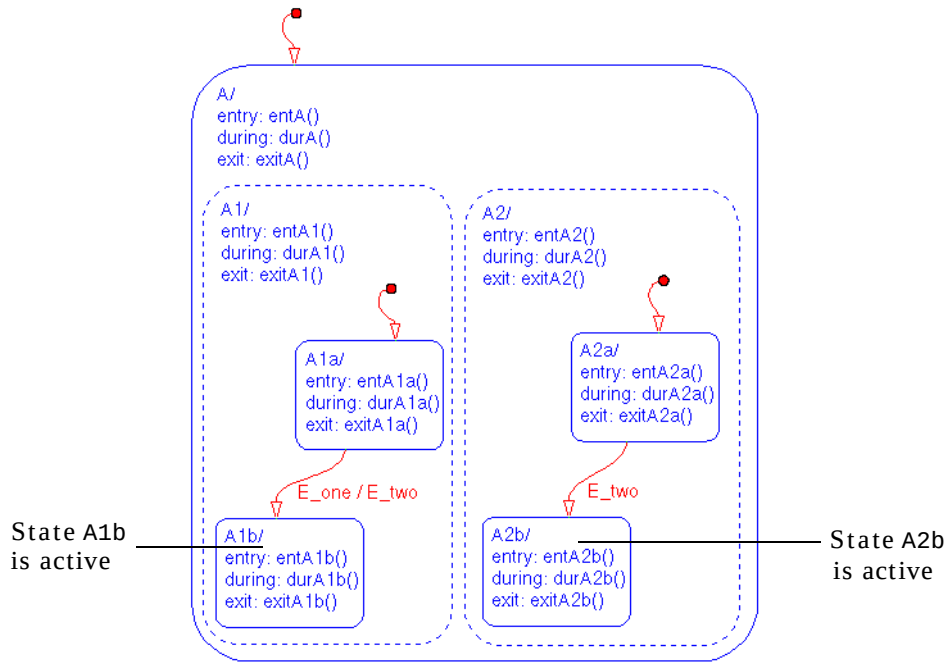
9 State A.A1.A1b is marked active.

10 Parallel state A.A2 is evaluated next. State A.A2 `during` actions execute and complete (`durA2()`). There are no valid transitions as a result of `E_one`.

11 State A.A2.A2b, now active as a result of the processing of the transition action event broadcast of `E_two`, executes and completes `during` actions (`durA2b()`).

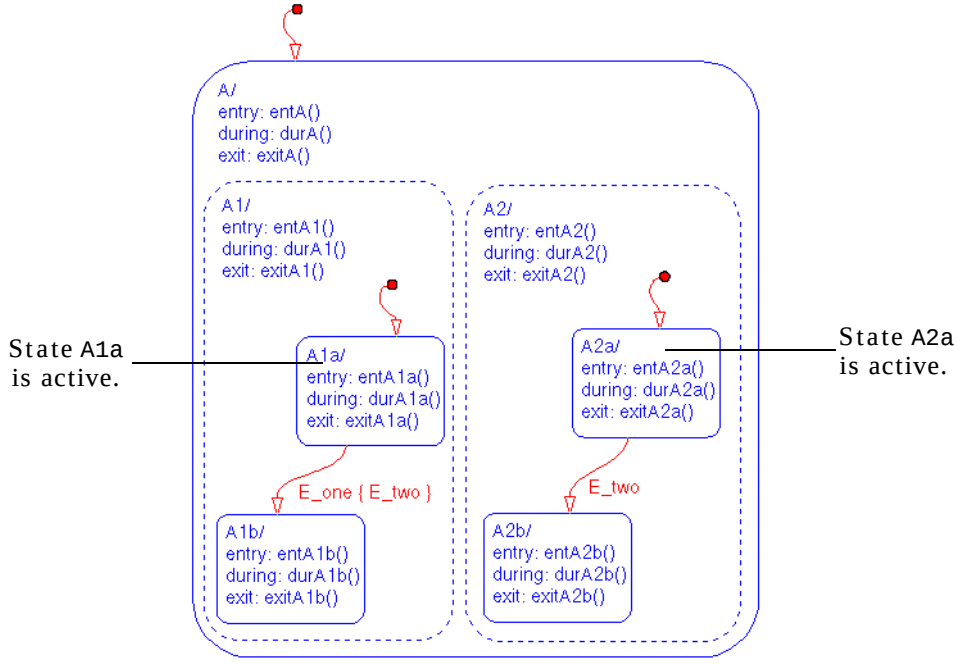
12 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

This sequence completes the execution of this Stateflow diagram associated with event `E_one` and the transition action event broadcast to a parallel state of event `E_two`. The final Stateflow diagram activity now looks like this.



Example: Event Broadcast Condition Action

This example shows the semantics of condition action event broadcast in parallel (AND) states.



Initially the Stateflow diagram is asleep. Parallel substates `A.A1.A1a` and `A.A2.A2a` are active. Event `E_one` occurs and awakens the Stateflow diagram. Event `E_one` is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of `E_one`. There is no valid transition.

2. State `A` executes and completes `during` actions (`durA()`).

3. State A's children are parallel (AND) states. Parallel states are evaluated and executed from top to bottom. In the case of a tie, they are evaluated from

left to right. State A.A1 is evaluated first. State A.A1 executes and completes during actions (durA1( )).

4 State A.A1 checks for any valid transitions as a result of event E_one. There is a valid transition from state A.A1.A1a to state A.A1.A1b. There is also a valid condition action. The condition action event broadcast of E_two is executed and completed. State A.A1.A1a is still active.

   a The broadcast of event E_two awakens the Stateflow diagram a second time. The Stateflow diagram root checks to see if there is a valid transition as a result of E_two. There is no valid transition.

   b State A executes and completes during actions (durA( )).

   c State A' s children are evaluated starting with state A.A1. State A.A1 executes and completes during actions (durA1( )). State A.A1 is evaluated for

valid transitions. There are no valid transitions as a result of `E_two` within state `A1`.

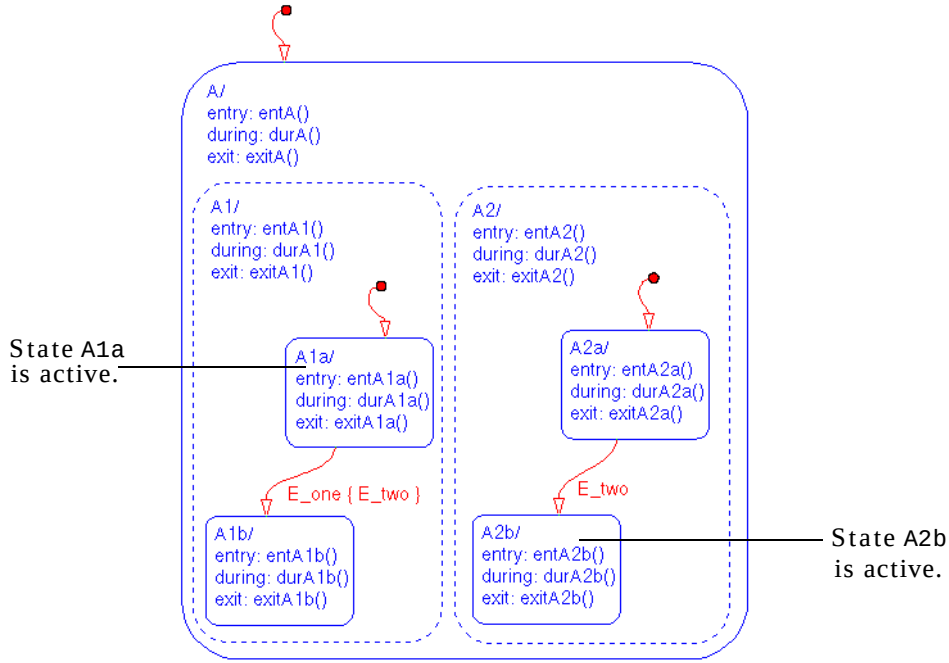
d State `A.A2` is evaluated. State `A.A2` executes and completes `during` actions (`durA2()`). State `A.A2` checks for valid transitions. State `A.A2` has a valid transition as a result of `E_two` from state `A.A2.A2a` to state `A.A2.A2b`.

e State `A.A2.A2a` `exit` actions execute and complete (`exitA2a()`).

f State `A.A2.A2a` is marked inactive.

g State `A.A2.A2b` is marked active.

h State `A.A2.A2b` `entry` actions execute and complete (`entA2b()`). The Stateflow diagram activity now looks like this.



5 State `A.A1.A1a` executes and completes `exit` actions (`exitA1a`).

6 State `A.A1.A1a` is marked inactive.

7 State A.A1.A1b executes and completes `entry` actions (`entA1b()`).

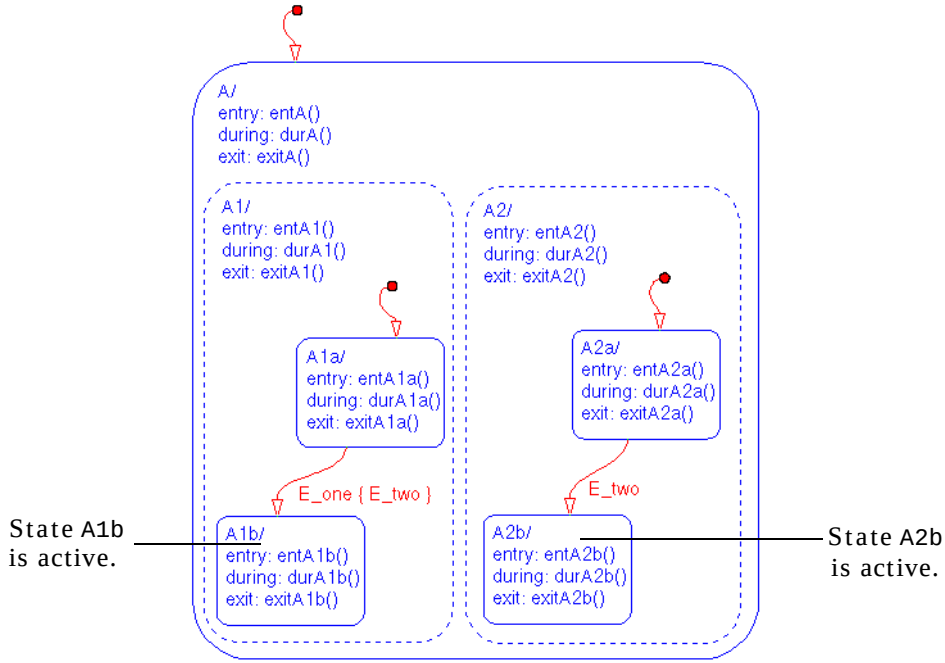
8 State A.A1.A1b is marked active.

9 Parallel state A.A2 is evaluated next. State A.A2 `during` actions execute and complete (`durA2()`). There are no valid transitions as a result of `E_one`.

10 State A.A2.A2b, now active as a result of the processing of the condition action event broadcast of `E_two`, executes and completes `during` actions (`durA2b()`).

11 The Stateflow diagram goes back to sleep waiting to be awakened by another event.

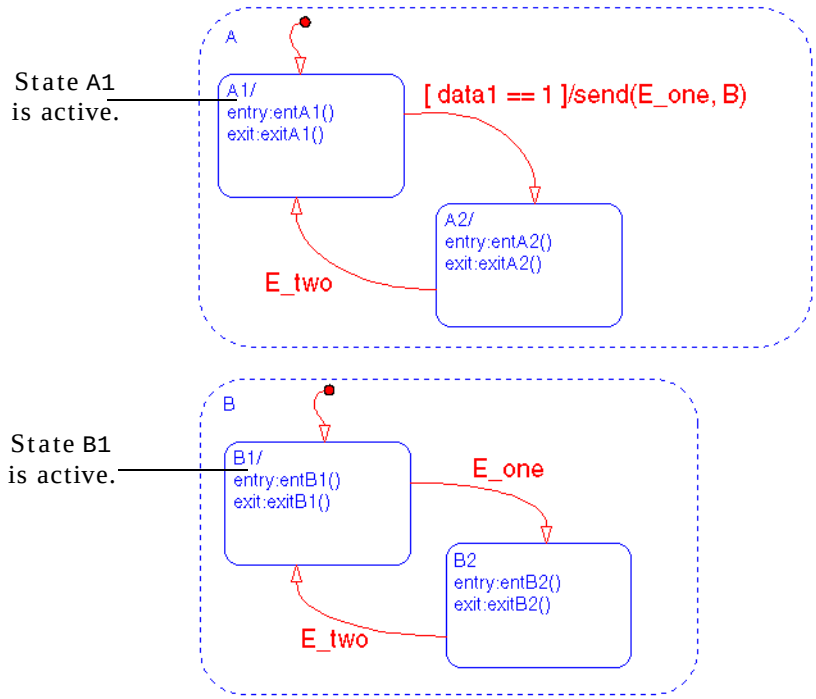
This sequence completes the execution of this Stateflow diagram associated with event `E_one` and the condition action event broadcast to a parallel state of event `E_two`. The final Stateflow diagram activity now looks like this.



Directed Event Broadcasting

Example: Directed Event Broadcast Using send

This example shows the semantics of directed event broadcast using `send(event_name,state_name)` in a transition action.



Initially the Stateflow diagram is asleep. Parallel substates `A.A1` and `B.B1` are active. By definition, this implies parallel (AND) superstates `A` and `B` are active. An event occurs and awakens the Stateflow diagram. The condition

[data1==1] is true. The event is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of the event. There is no valid transition.
2. State A's children are parallel (AND) states. They are evaluated and executed from left to right and top to bottom. State A.A1 is evaluated first.
3. State A.A1 checks for any valid transitions as a result of the event. Since the condition [data1==1] is true, there is a valid transition from state A.A1 to state A.A2.
4. State A.A1 exit actions execute and complete (exitA1()).

Start of event E_one processing.

5. State A.A1 is marked inactive.
6. The transition action, send(E_one,B) is executed and completed.
   a. The broadcast of event E_one awakens state B. (This is a nested event broadcast.) Since state B is active, the directed broadcast is received and state B checks to see if there is a valid transition. There is a valid transition from B.B1 to B.B2.
   b. State B.B1 executes and completes exit actions (exitB1()).
   c. State B.B1 is marked inactive.
   d. State B.B2 is marked active.

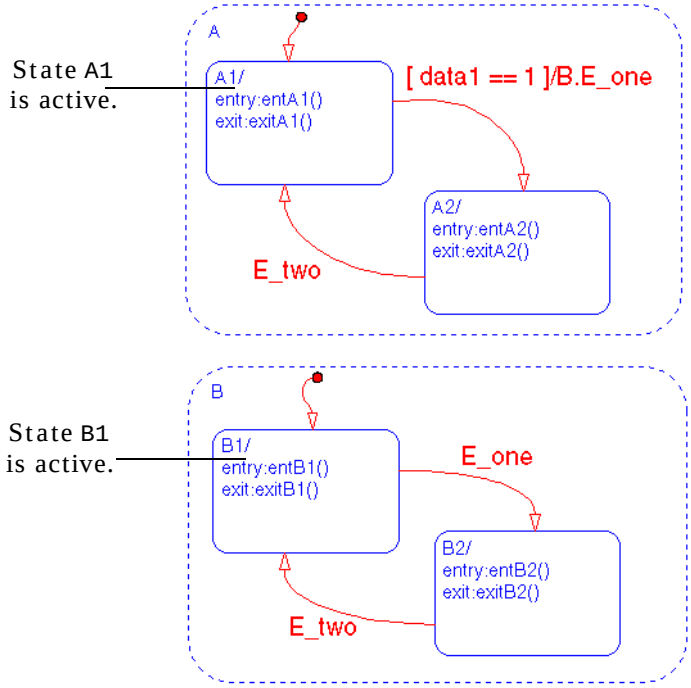
End of event E_one processing.

7. State A.A2 is marked active.
8. State A.A2 entry actions execute and complete (entA2()).

This sequence completes the execution of this Stateflow diagram associated with an event broadcast and the directed event broadcast to a parallel state of event E_one.

Example: Directed Event Broadcasting Using Qualified Event Names

This example shows the semantics of directed event broadcast using a qualified event name in a transition action.



Initially the Stateflow diagram is asleep. Parallel substates A.A1 and B.B1 are active. By definition, this implies parallel (AND) superstates A and B are active. An event occurs and awakens the Stateflow diagram. The condition

[data1==1] is true. The event is processed from the root of the Stateflow diagram down through the hierarchy of the Stateflow diagram.

1. The Stateflow diagram root checks to see if there is a valid transition as a result of the event. There is no valid transition.

2. State A's children are parallel (AND) states. They are evaluated and executed from left to right and top to bottom. State A.A1 is evaluated first.

3. State A.A1 checks for any valid transitions as a result of the event. Since the condition [data1==1] is true, there is a valid transition from state A.A1 to state A.A2.

4. State A.A1 exit actions execute and complete (exitA1()).

5. State A.A1 is marked inactive.

6. The transition action, a qualified event broadcast of event E_one to state B (represented by the notation B.E_one), is executed and completed.

   a. The broadcast of event E_one awakens state B. (This is a nested event broadcast.) Since state B is active, the directed broadcast is received and state B checks to see if there is a valid transition. There is a valid transition from B.B1 to B.B2.

   b. State B.B1 executes and completes exit actions (exitB1()).

   c. State B.B1 is marked inactive.

   d. State B.B2 is marked active.

7. State A.A2 is marked active.

8. State A.A2 entry actions execute and complete (entA2()).

This sequence completes the execution of this Stateflow diagram associated with an event broadcast using a qualified event name to a parallel state.

Execution Order

Overview

In a single processor environment, sequential execution order is the only option. In this case, it may be necessary for you to know the implicit ordering determined by a Stateflow diagram. The ordering is specific to transitions originating from the same source. Knowing the order of execution for Stateflow diagrams with more than one parallel (AND) state may be important.

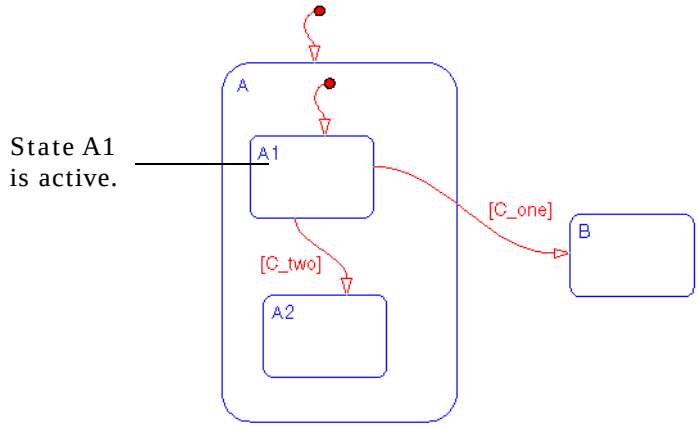
We recommend that you do not design your Stateflow diagram based on an expected execution order.

Execution Order Guidelines

Execution order of transitions originating from the same source is based on these guidelines. The guidelines appear in order of their precedence.

1 Transitions are evaluated, based on hierarchy, in a top-down manner. In this example, when an event occurs and state A.A1 is active, the transition

from state A.A1 to state B is valid and takes precedence over the transition from state A.A1 to state A.A2 based on the hierarchy.



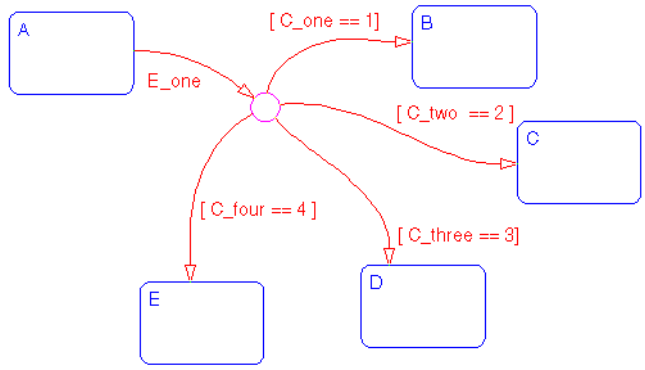
2 Transitions are evaluated based on their labels.

- **a** Labels with events and conditions
- **b** Labels with events
- **c** Labels with conditions
- **d** No label

3 Equivalent transitions (based on their labels) are evaluated based on the geometry of the outgoing transitions. The geometry of junctions and states is considered separately.

Junctions

Multiple outgoing transitions from junctions that are of equivalent label priority are evaluated in a clockwise progression starting from a twelve o'clock position on the junction.



In this example, the transitions are of equivalent label priority. The conditions [C_three == 3]and [C_four == 4] are both true. Given that, the outgoing transitions from the junction are evaluated in this order:

1 A -> B

Since the condition [C_one == 1] is false, this transition is not valid.

2 A -> C

Since the condition [C_two == 2] is false, this transition is not valid.

3 A -> D

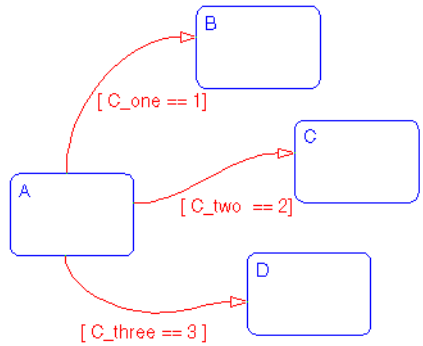
Since the condition [C_three == 3] is true, this transition is valid and is taken.

4 A -> E

This transition, even though it too is valid, is not evaluated since the previous transition evaluated was valid.

States

Multiple outgoing transitions from states that are of equivalent label priority are evaluated in a clockwise progression starting from a twevle o'clock position on the state.



In this example, the transitions are of equivalent label priority. The conditions [C_two == 2] and [C_three == 3] are both true and [C_one = = 1] is false. Given that, the outgoing transitions from the state are evaluated in this order:

1 A -> B

Since the condition [C_one == 1] is false, this transition is not valid.

2 A -> C

Since the condition [C_two == 2] is true, this transition is valid and is taken.

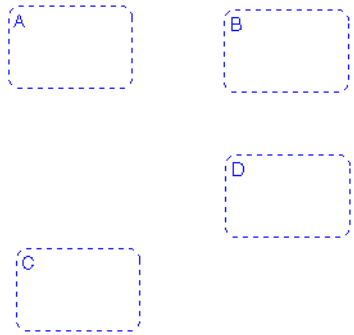
3 A -> D

This transition, even though it too is valid, is not evaluated since the previous transition evaluated was valid.

Parallel (AND) States

Parallel (AND) states are evaluated and executed first from top to bottom and then from left to right in the case of a tie. In this example, assuming that A and

B, and C and D are exactly equivalent from top-down, the parallel (AND) states are executed in this order: A, B, D, C.



Semantic Rules Summary

Event Broadcasting Rules

In summary, these are the rules for event broadcasting:

- A Stateflow diagram is event driven; the Stateflow diagram awakens when an event is broadcast.
- An event broadcast occurs in a top-down manner starting from the parent of the event or the intended directed state.
- **Imported** and **Output to Simulink** events can be broadcast but cannot be received (cannot trigger transitions).

Rules for Processing States

In summary, these are the rules for processing states:

- The sole purpose of an active state is to become inactive through a valid state transition.
- Active states are processed from the top-down in a depth-first manner.
- Active exclusive (OR) states are processed:
  - **a** Outer transitions are processed in the search for a valid path from the state.
  - **b** Any state `during` actions (including `on` *`event_name`* actions) are executed.
  - **c** Inner transitions are processed in the search for a valid path within the state.
  - **d** If the state is a superstate with no valid transition paths and if the superstate remains active, then the processing continues in (is passed down to) active substate(s).
- Active parallel (AND) states are processed:
  - **a** Any `during` actions (including `on` *`event_name`* actions) of the state are executed.
  - **b** Inner transitions are processed in the search for a valid path within the state.
  - **c** If the state is a superstate with no valid transition paths and if the superstate remains active, then the processing continues in (is passed down to) active substate(s).
- Note that in each of the above steps, when a successful transition path is found, it is taken. The prerequisite for each step is that the exclusive (OR) state is still active. If for any reason the exclusive (OR) state becomes

inactive, its processing is regarded as complete and no further steps are taken.

- When the execution of a Stateflow diagram is completed, it must be in a consistent state.
  - An active state (consisting of at least one substate) with XOR decomposition has exactly one active substate.
  - All substates of an active state with AND decomposition are active.
  - All substates of an inactive state with either XOR or AND decomposition are inactive.
- The order of execution of parallel (AND) states in a superstate with parallel (AND) decomposition is determined by the geometrical sorting of states from top to bottom and then from left to right in the case of a tie.

Rules for Processing Transitions

In summary, these are the rules for processing transitions:

- State transitions are achieved through taking valid transition paths.
- A valid transition path is found through the process of considering all candidate paths, segment by segment, in precise, deterministic order.
- For a path to be valid, all transition segments along the path must be valid.
- With the first invalid segment, a candidate path is rejected and not processed further.
- While processing for a valid path, condition actions of valid transition segments are executed.
- Transitions are processed at a decision point in this order:
  - **a** Transitions with higher level parents
  - **b** Event triggered + guarding condition
  - **c** Event triggered only
  - **d** Guarding condition only
  - **e** Unconditional
  - **f** Clockwise twelve o'clock rule for junctions
  - **g** Clockwise from top-left corner rule for states

Rules When Taking Transition Paths

In summary, these are the rules when taking transition paths:

- The parent of a valid transition path is the innermost state that contains all the points along the transition path.
- All of the active states up to the parent of the transition path are exited in a bottom-up hierarchical manner.
- When exiting a superstate with exclusive (OR) decomposition, first exit the active child and then exit the superstate.
- When exiting a superstate with parallel (AND) decomposition, first exit all active children in a deterministic order (geometrically from right to left and from the bottom of to the top of the Stateflow diagram) and then exit the superstate. (This is the reverse of entry into such superstates.)
- Just before exiting each state, the state's `exit` action is executed. Note that when executing the `exit` action, the state is still marked active. A state exit is complete when it is marked inactive.
- After exiting all active states along the transition path, the transition actions of the path are executed in the order they appear along the path.
- At this point the transition into new states is executed in a top-down hierarchical manner.
- When entering an exclusive (OR) state, its superstate with exclusive (OR) decomposition is entered first.
- When entering a parallel (AND) state, its superstate with parallel (AND) decomposition is also entered. Note that all siblings of this parallel (AND) state must also be entered. These children are entered in a deterministic order (geometrically from left to right and from the top to the bottom of the Stateflow diagram). (This is the reverse of exit from such superstates.)
- Just after entering each state, its `entry` action is executed. Note when executing the `entry` action, the state is already marked active.
- When the transition path terminates on the edges of a superstate, the entry into its substate(s) is determined by processing the default transition paths of the superstate.
- When the transition path crosses the edges of a superstate to terminate on a substate, the entry into its substate(s) is controlled by the transition path explicitly. The default paths of the superstate are not processed.

8

Tutorial: Entering a Stateflow Diagram

Overview 8-2

Designing the Carousel Stateflow Diagram 8-4

Procedure 8-8

Run the Simulation for the Provided Solution 8-24

Run the Carousel Demo 8-27

Overview

Objectives

After completing this tutorial you should be able to:

- Understand the Stateflow diagram design process for the carousel.
- Understand the notation and methods to create a model and Stateflow diagram using Stateflow.
- Create an initial model and Stateflow diagram of the carousel.
- Run the simulation using the provided solution.
- Run the demo of the solution.

Provided Models for This Chapter

The files associated with the tutorial example, the running solution, and the demo are included and accessible from the Examples block.

Carousel Tutorial Example

A carousel is a familiar reactive system that you can model with a Stateflow diagram. The system reacts based upon the operator's actions.

This tutorial focuses on learning the notation and the methods to create an initial Stateflow diagram for the carousel. The completed tutorial example cannot be simulated. The provided solution includes a Simulink model and can be simulated. The demo is more elaborate and shows how the Simulink model can be combined with a GUI.

Basic Operation

The carousel is divided into these main physical components:

- Main platform of the ride
- Audio
- Video

The carousel can either be on or off. When the carousel is on, the platform can be stopped or moving, depending on whether a ride is in progress. You have the option of having music play and displaying a video image of the carousel that

rotates on the operator's console. You enable audio and video before starting the ride. When the operator of the ride pushes the start button, the platform turns (music plays and video rotates if enabled). The ride lasts the amount of time the operator specifies.

If the carousel exceeds a maximum speed, it stops automatically. There is an emergency stop button on the carousel operator's console if, for any reason, the operator decides the ride needs to be stopped.

Designing the Carousel Stateflow Diagram

Overview

A Stateflow diagram is only as good as the up-front analysis and design work for the desired system behavior. Before you can create your Stateflow diagram, you need to interpret the verbal description of the system behavior and translate that description into Stateflow notation. For the purposes of this tutorial, some aspects of the operation are simplified.

The Design Questions

Addressing these questions helps in creating a Stateflow diagram that is representative of the carousel system:

- What are the states?
- What is the decomposition of the states?
- What should the state labels be?
- Are there junctions?
- What are the transitions?
- Are there default transitions?
- Are there events?
- Is there data?

What Are the States?

Begin by listing possible states in the carousel system. A good place to begin is by considering the physical components of the carousel. There are likely to be states associated in some way with some of the physical components.

Audio

The audio components are a subset of the carousel system as a whole. If the audio system is enabled on the operator's console, music plays when the ride is in progress. In this tutorial, the substates of the audio component are not discussed in any detail.

Video

The video components are a subset of the carousel system as a whole. If the video system is enabled on the operator's console, an image of a rotating platform is displayed on the operator's console. In this tutorial, the substates of the video component are not discussed in any detail.

Ride

The platform movement is the ride. There is a state associated with the ride as a whole. In the simplest sense the ride is either in progress or not. There is one state, Ride, that consists of two substates, `On`, and `Stopped`.

Timer

A state for a timer is somewhat less obvious than the others. The duration of the ride is set by the operator. It is useful to define a separate state for the timer that will generate the event indicating when the ride is done. The timer is either counting or stopped. There are three states associated with the timer: `Counting`, `Stopped`, and `Timer`.

The Carousel System

The carousel system is either on or off. There are two states associated with the carousel at the system level: `Power_on` and `Power_off`.

What Is a State Decomposition?

The Stateflow diagram level of the hierarchy has exclusive (OR) decomposition and consists of two OR states: `Power_on` and `Power_off`.

The `Power_on` state has parallel (AND) decomposition and consists of these AND substates: `Timer`, `Audio`, `Video`, and `Ride`. When the `Power_on` state is active, the `Timer`, `Audio`, `Video`, and `Ride` states are active. They are essentially subsystems of the carousel system.

The `Timer`, `Ride`, `Audio`, and `Video` states have exclusive (OR) decomposition.

See “What Is a State Decomposition?” on page 6-10 for more information.

What Are the State Labels?

Part of the process of defining each state is defining the state label. The state name is the part of the label that describes the state in some recognizable way. The state labels in the carousel consist of the name of the state.

See the section titled “Labeling a State” on page 6-13 for information on valid state labels.

Will Junctions Simplify the Design?

The carousel system as described and designed does not have junctions.

See the sections titled “Connective Junctions” on page 6-32 and “History” on page 6-40 for information on the use and notation for Connective and History junctions.

What Are the Transitions?

In the carousel system, these are the transitions between OR states:

- `Power_off`→ `Power_on` triggered by event `Switch_on`
- `Power_on`→`Power_off` triggered by event `Switch_off`
- `Power_on.Ride.On`→`Power_on.Ride.Stopped` triggered by any event when the condition `[Speed > Max_speed]` is true
- `Power_on.Ride.On`→`Power_on.Ride.Stopped` triggered by event `Emergency_stop`
- `Power_on.Ride.On`→`Power_on.Ride.Stopped` triggered by event `Ride_done`
- `Power_on.Ride.Stopped`→`Power_on.Ride.On` triggered by event `Start_button`
- `Power_on.Timer.Stopped`→`Power_on.Timer.Counting` triggered by event `Start_button`
- `Power_on.Timer.Counting`→`Power_on.Timer.Stopped` triggered by any event when the condition `[Time > Duration]` is true. The event `Ride_done` is broadcast as a transition action.

See the section titled “Transitions” on page 6-17 for more information.

Are There Default Transitions?

These are the default transitions in the example:

- The `Power_off` substate is the default state to become active upon a transition to the carousel Stateflow diagram.
- The `Power_on.Ride.Stopped` substate is the default state to become active upon a transition to the `Ride` superstate.
- The `Power_on.Timer.Stopped` substate is the default state to become active upon a transition to the `Timer` superstate.

See the section titled “What is a Default Transition?” on page 6-24 for more information.

Are There Events?

These events are defined: `Ride_done`, `Switch_on`, `Switch_off`, `Start_button`, and `Emergency_stop`.

See the sections titled “Events” on page 3-5 for more information on events.

Is There Data?

The carousel system has these data objects: `Time`, `Speed`, `Duration`, and `Max_speed`.

See the section titled “Data” on page 3-22 for more information on data.

Procedure

Overview

A carousel is a familiar reactive system that you can model with a Stateflow diagram. The system reacts based upon actions the ride operator takes to use it. For the purposes of this tutorial, some aspects of the operation are simplified.

Use the sample solution in this section as a guide while following these steps. In this section you learn how to:

- Create a simple Simulink model with a Stateflow block.
- Save an untitled model and Stateflow diagram.
- Create and label the OR states.
- Create and label the AND states.
- Create and label the superstates.
- Create the transitions.
- Label the transitions.
- Create default transitions.
- Add and define event objects.
- Add and define data objects.
- Save the Stateflow diagram and exit.

Create a Simulink Model

These steps describe how to invoke Stateflow and save the untitled model and Stateflow diagram.

1. Double-click on the MATLAB icon to invoke MATLAB.

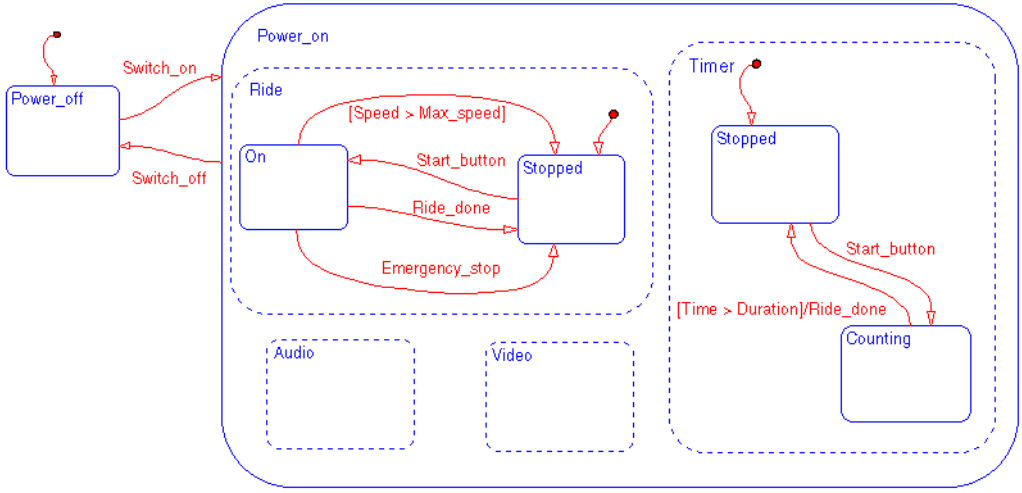
2. At the MATLAB prompt, type `sfnew` to display an untitled model window.

   Choose **Save As...** from the **File** menu of the model window. Enter a unique model name, modify the saved location if desired, and click on the **OK** button.

3. Click in the text area of the Stateflow block and enter a unique Stateflow block name. Double-click on the Stateflow block to invoke the Stateflow graphics editor.

Sample Stateflow Diagram

This is a sample of what the completed Stateflow diagram should look like.



Create and Label the OR States

1 Double-click on the **State** button in the toolbar. The button appears depressed and is now in multiple object mode. Using the sample solution Stateflow diagram as a guide, click in the drawing area to place the states in the drawing area. The Stateflow diagram is exclusive (OR) decomposition

by default, so you do not need to change the decomposition. The states correspond to these states in the solution:

- `Power_off`
- `Power_on.Ride.On`
- `Power_on.Ride.Stopped`
- `Power_on.Timer.Stopped`
- `Power_on.Timer.Counting`

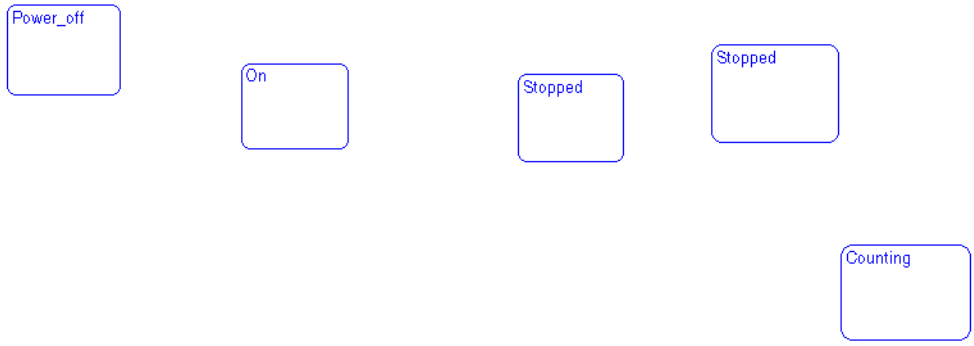
Press the **Escape** key or click on the **State** button to toggle the button out of multiple object mode.

2 Click on the ? character within each state to enter each state label. The final text after the last period is the individual state's name. There are some states that have the same text as their label. This is ok since the hierarchy of the state is used to create a unique label. For example, both the `Ride` and `Timer` superstates have a `Stopped` substate. Reading the Stateflow diagram, you can see that these states are uniquely described by the text `Power_on.Ride.Stopped` and `Power_on.Timer.Stopped`.

Use the sample solution to label the states based on their placement in the drawing area. Deselect the state (press the **Escape** key) to exit the edit.

Sample Snapshot

Your Stateflow diagram should look similar to this Stateflow diagram.


Power_off
On
Stopped
Stopped
Counting

Create and Label the Superstates

To create a superstate, create a state and resize so that it includes or surrounds the desired substates. These steps show how to create and label the superstates.

1. Click on the **State** button from the toolbar. Using the sample solution as a guide, place the `Ride` superstate in the drawing area.

2. Resize the state to create the superstate. Dragging any one of the state's corners resizes the state. You know you are at the resizing location on the state border when the cursor appears as a double-ended arrow.

3. Click on the ? within the superstate and enter the text `Ride`. Deselect the superstate to exit the edit.

4. Click on the **State** button and place the `Power_on` state in the drawing area.

5. Resize the `Power_on` state to create a superstate. Dragging any one of the state's corners resizes the state. You know you are at the resizing location on the state border when the cursor appears as a double-ended arrow.

6. Click on the ? within the OR superstate and enter the text `Power_on`. Deselect the OR superstate to exit the edit.

7. Click on the **State** button and place the `Timer` state in the drawing area.

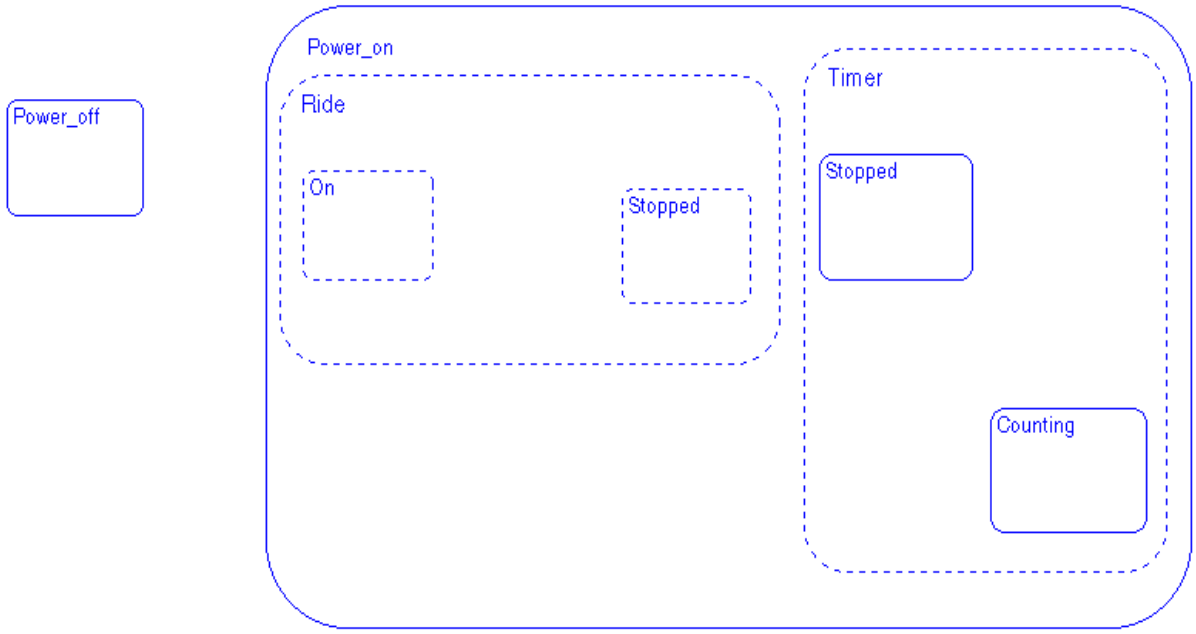
8. Resize the `Timer` state to create a superstate. Dragging any one of the state's corners resizes the state. You know you are at the resizing location on the state border when the cursor appears as a double-ended arrow.

9. Click on the ? within the OR superstate and enter the text `Timer`. Deselect the OR superstate to exit the edit.

10. Display the state shortcut menu by placing the cursor inside the borders of the `Power_on` state but outside the borders of any of `Power_on`'s substates and clicking the right mouse button. Choose the **Parallel (AND) decomposition** menu item.

Sample Snapshot

Your Stateflow diagram should now look similar to this Stateflow diagram.



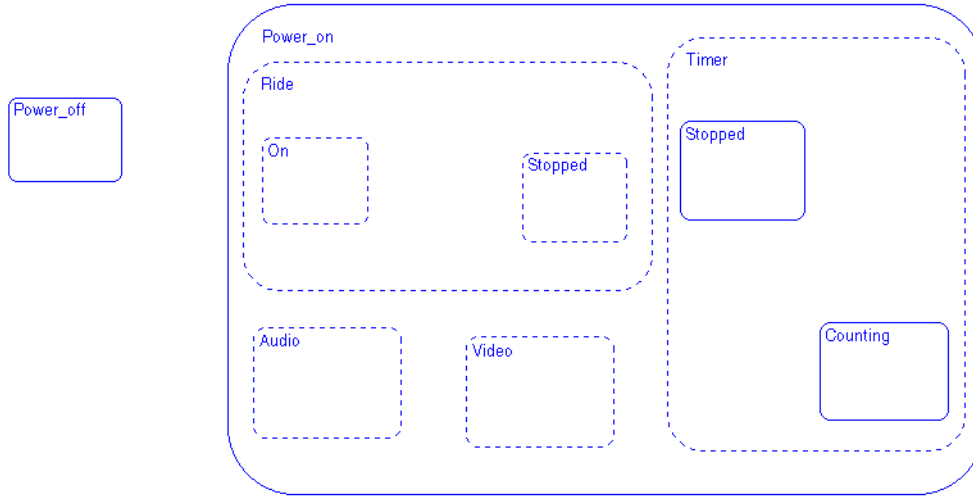
Create and Label the AND States

1 Double-click on the **State** button in the toolbar. The button appears depressed and is now in multiple object mode. Using the sample solution Stateflow diagram as a guide, click in the drawing area to place the `Audio` and `Video` states in the drawing area. The states are created as AND states since the decomposition of `Ride` is already set. Press the **Escape** key or click on the **State** button to toggle the button out of multiple object mode.

2 Click on the ? character within each state to enter each state label. Use the sample solution to label the states based on their placement in the drawing area. Deselect the state (press the **Escape** key) to exit the edit.

Sample Snapshot

Your Stateflow diagram should look similar to this Stateflow diagram.



Create the Transitions

These steps show how to create the transitions.

1. Start with the transitions within the `Ride` superstate. Using the solution as a guide, place the cursor at a straight portion of the side border of the `Power_on.Ride.Stopped` substate. Click the border when the cursor changes to a cross-hair. Without releasing the mouse button, drag the mouse to a straight portion on the side border of the `Power_on.Ride.On` substate. When the transition snaps to the border, release the mouse button. Transitions cannot be attached on corners since corners are used for resizing. You create one transition in this step.

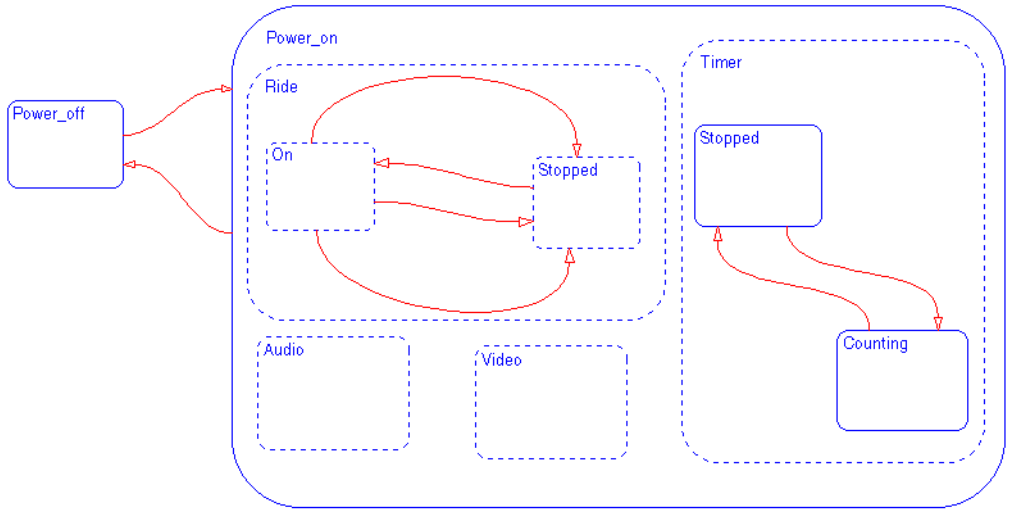
2. Create the rest of the transitions within the `Ride` superstate. You create three transitions in this step.

3. Create the transitions within the `Timer` superstate. You create two transitions in this step.

4. Create the transitions to and from the `Power_off` state. You create two transitions in this step.

Sample Snapshot

Your Stateflow diagram should now look similar to this Stateflow diagram.



Label the Transitions

The general format of a transition label is:

```
event[condition]{condition_action}/transition_action
```

In this example, there are eight transitions to label. Use the transition labels in the solution Stateflow diagram as a guide. These steps show how to label the transitions.

1 Start with the transition from the substate `Power_on.Ride.Stopped` to the substate `Power_on.Ride.On`. Select the transition. Click on the ? alongside the transition and enter the label `Start_button`. Deselect the transition label (press the **Escape** key) to exit the edit. You label one transition in this step. To move a transition label, click and drag the left mouse button to the new desired location and then release the mouse button. If you mistakenly

click and release the left mouse button, you will be in label edit mode. Deselect the label and try again.

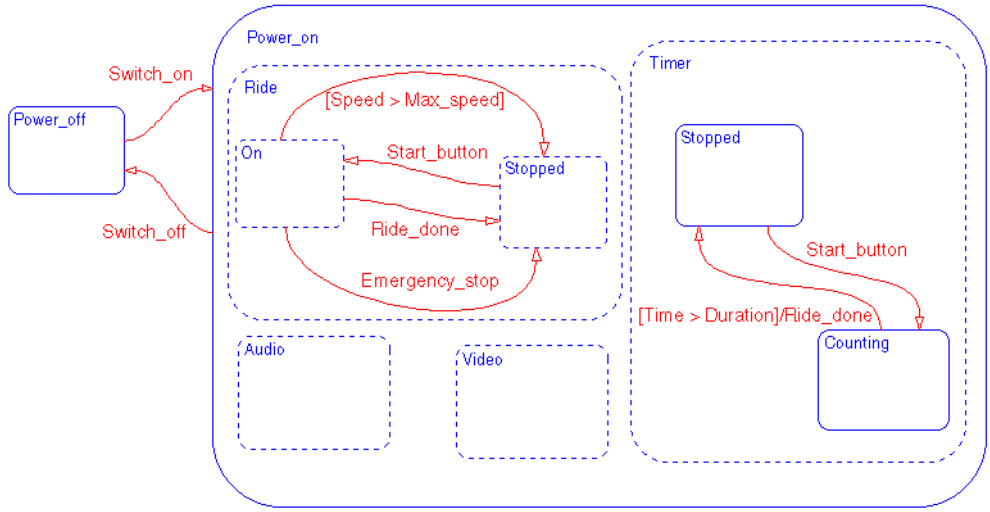
2 Label the remaining transitions in the `Power_on.Ride` superstate. You label three transitions in this step.

3 Label the transitions in the `Timer` superstates. You label two transitions in this step.

4 Label the transitions to and from the `Power_off` substate. You label two transitions in this step.

Sample Snapshot

Your Stateflow diagram should now look similar to this Stateflow diagram.



Create Default Transitions

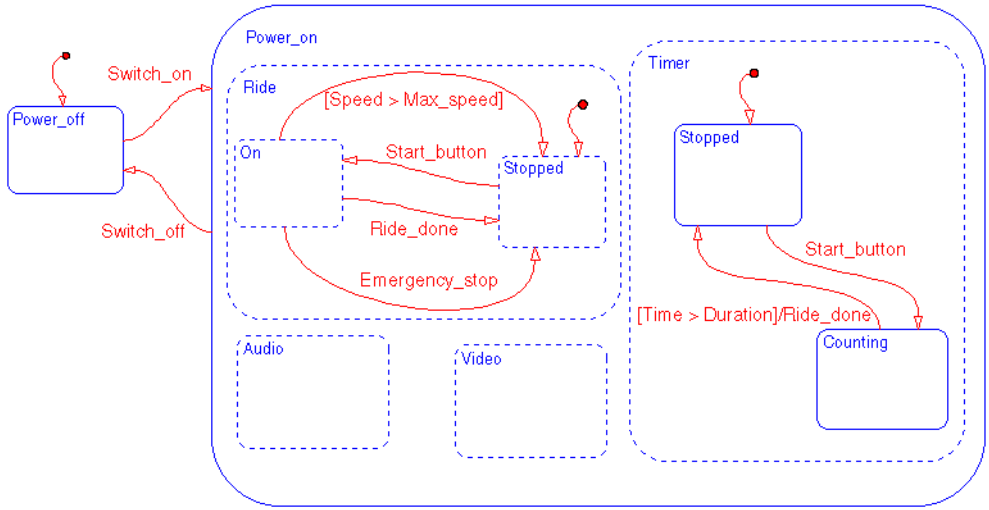
There are three Default transitions in the solution Stateflow diagram.

Double-click on the **Default transition** button in the toolbar. The button appears depressed and is now in multiple object mode. Using the sample solution as a guide, place the default transitions in the drawing area.

This is the final step in creating the graphic objects in the Stateflow diagram.

Sample Snapshot

Your Stateflow diagram should now look similar to this Stateflow diagram.



Add and Define Event Objects

Events are added, removed, and edited using the Explorer. These events must be defined: `Ride_done`, `Switch_on`, `Switch_off`, `Start_button`, `Emergency_stop`.

Add and Define a Local Event

`Ride_done` is the only local event in the carousel example.

1. Choose **Explore** from the **Tools** menu of the graphics editor window to display the Explorer.

2. Select the parent for the new event.

   Double-click on the machine name (the name you specified when you saved the Simulink model) in the **Object Hierarchy** list. Double-click on the chart

name (the name you specified on the Stateflow block). The chart will be the parent for all the events in the carousel example.

3 Choose **Event** from the Explorer **Add** menu. The **Event properties** dialog box is displayed.

4 Enter `Ride_done` in the name field.

5 Select **Local** in the **Scope** field.

6 Click on the **Close** button to save the changes and close the dialog box.

Add and Define Input from Simulink Events

There are four **Input from Simulink** events to add and define: `Switch_on`, `Switch_off`, `Start_button`, and `Emergency_stop`.

1 Choose **Explore** from the **Tools** menu of the graphics editor window to display the Explorer.

2 Select the parent for the new event.

Double-click on the machine name (the name you specified when you saved the Simulink model) in the **Object Hierarchy** list. Double-click on the chart

name (the name you specified on the Stateflow block). The chart will be the parent for all the events in the carousel example.

3 Choose **Event** from the Explorer **Add** menu. The **Event properties** dialog box is displayed.

4 Enter `Switch_on` in the name field.

5 Select **Input from Simulink** in the **Scope** field.

6 Click on the **Apply** button to save the **Scope** field.

7 Since the **Scope** is **Input from Simulink**, an index number is displayed. You do not need to change the index number.

8 Select **Either Edge** in the **Trigger** field.

9 Click on the **Close** button to save the changes and close the dialog box.

10 Repeat these steps for each **Input from Simulink** event, entering the appropriate text in the **Name** field.

Add and Define Data Objects

Data are added, removed, and edited using the Explorer. These data objects must be defined: `Time`, `Speed`, `Duration`, and `Max_speed`.

Add and Define Local Data

The `Time` data object is local.

1 Choose **Explore...** from the **Tools** menu of the graphics editor.

2 Select the parent for the new data object.

Double-click on the machine name (the name you specified when you saved the Simulink model) in the **Object Hierarchy** list. Double-click on the chart

name (the name you specified on the Stateflow block). The chart will be the parent for all the data in the carousel example.

3 Choose **Data** from the Explorer **Add** menu. The **Data properties** dialog box is displayed.

4 Enter `Time` in the **Name** field.

5 Select **Local** in the **Scope** field.

6 Click on the **Close** button to save the changes and close the dialog box.

Add and Define Input from Simulink Data

The `Speed` data object is **Input from Simulink**.

1 Choose **Explore...** from the **Tools** menu of the graphics editor.

2 Select the parent for the new data object.

Double-click on the machine name (the name you specified when you saved the Simulink model) in the **Object Hierarchy** list. Double-click on the chart name (the name you specified on the Stateflow block). The chart will be the parent for all the data in the carousel example.

3 Choose **Data** from the Explorer **Add** menu. The **Data properties** dialog box is displayed.

4 Enter `Speed` in the **Name** field.

5 Select **Input from Simulink** in the **Scope** field.

6 Since the **Scope** is **Input from Simulink**, a port number is displayed. You do not need to change the port number.

7 Click on the **Close** button to save the changes and close the dialog box.

Add and Define Data Output to Simulink

`Ride_on` is the only data **Output to Simulink**.

1. Choose **Explore...** from the **Tools** menu of the graphics editor.
2. Select the parent for the new data object.

   Double-click on the machine name (the name you specified when you saved the Simulink model) in the Object Hierarchy list. Double-click on the chart name (the name you specified on the Stateflow block). The chart will be the parent for all the data in the carousel example.
3. Choose **New** and then **Data** from the Explorer **File** menu. The **Data object properties** dialog box is displayed.
4. Enter `Ride_on` in the **Name** field.
5. Select **Output to Simulink** in the **Scope** field.
6. Since the **Scope** is **Output to Simulink**, a port number is displayed. You do not need to change the port number.
7. Click on the **Close** button to save the changes and close the dialog box.

Add and Define Workspace Data

`Duration` is the only **Workspace** data object.

1. Choose **Explore...** from the **Tools** menu of the graphics editor.
2. Select the parent for the new data object.

   Double-click on the machine name (the name you specified when you saved the Simulink model) in the Object Hierarchy list. Double-click on the chart

name (the name you specified on the Stateflow block). The chart will be the parent for all the data in the carousel example.

3 Choose **New** and then **Data** from the Explorer **File** menu. The **Data object properties** dialog box is displayed.

4 Enter `Duration` in the **Name** field.

5 Select **Workspace** in the **Scope** field.

6 Click on the **Close** button to save the changes and close the dialog box.

Add and Define Constant Data

`Max_speed` is the only **Constant** data object.

1 Choose **Explore...** from the **Tools** menu of the graphics editor.

2 Select the parent for the new data object.

Double-click on the machine name (the name you specified when you saved the Simulink model) in the Object Hierarchy list. Double-click on the chart name (the name you specified on the Stateflow block). The chart will be the parent for all the data in the carousel example.

3 Choose **New** and then **Data** from the Explorer **File** menu. The **Data object properties** dialog box is displayed.

4 Enter `Max_speed` in the **Name** field.

5 Select **Constant** in the **Scope** field.

6 Click on the **Close** button to save the changes and close the dialog box.

Save the Model

1 Choose **Save** from the graphics editor **File** menu to save the entire model.

2 If you want to exit from Stateflow and MATLAB, choose **Exit MATLAB** from the **File** menu of the Stateflow graphics editor.

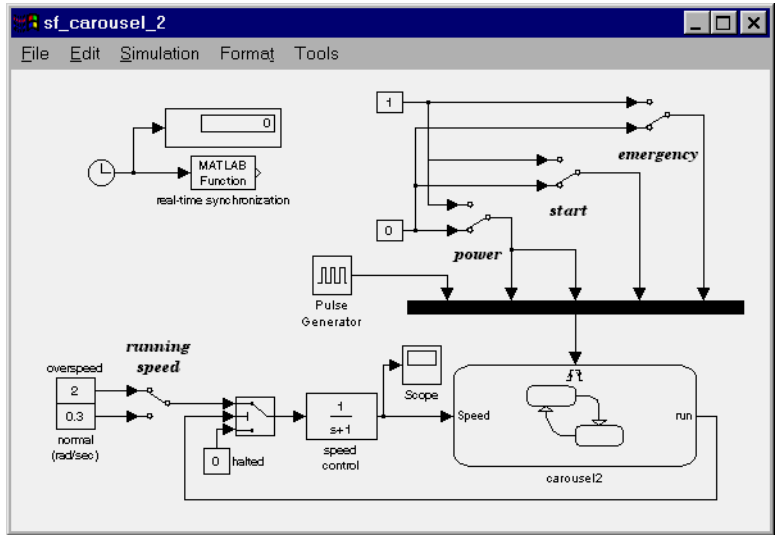
Run the Simulation for the Provided Solution

Overview

The Simulink model and Stateflow diagram created in "Procedure" on page 8-8 is not complete enough to run a simulation. This provided solution is based on the tutorial and extends it.

Carousel Simulink Model Solution

This is the provided carousel Simulink model solution:



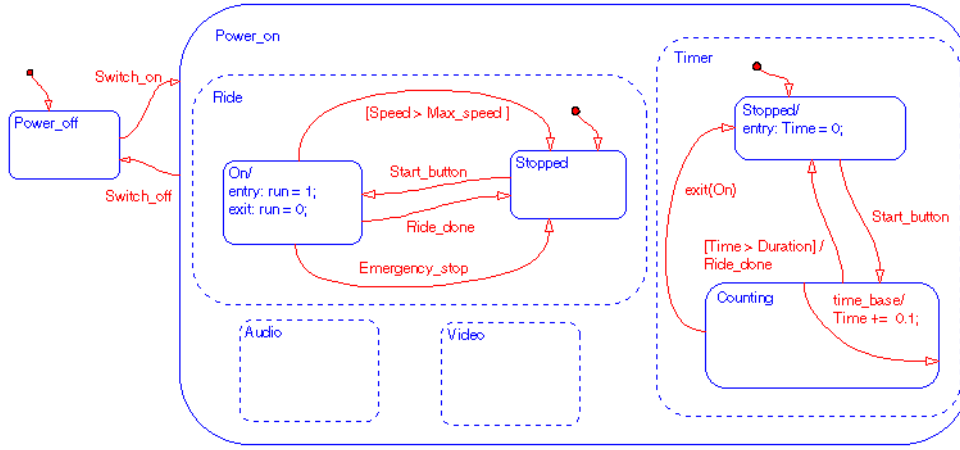
Carousel Operator's Console

The switches in the Simulink model are used to act as a carousel operator's console. The switches allow you to:

- Turn the carousel power on and off
- Start the ride
- Stop the ride in case of emergency
- Specify whether the speed is in the normal or overspeed range

Carousel Solution Stateflow Diagram

This is the provided carousel Stateflow diagram solution:



Procedure

Follow these steps to run the simulation:

1. Double-click on the Examples block in the Stateflow model window. Double-click on the Carousel and then the Basic Functional Carousel block.

2. Choose **Start** from the graphics editor **Simulation** menu.

   The first time you run the simulation, and any time you have modified the Stateflow diagram, choosing **Start** will parse the machine, generate simulation code and generate the simulation target. If the target build takes

place, informational messages are displayed in popup windows and in the MATLAB command window.

In any case, the graphics editor is 'ICED' and modifications to the Stateflow diagram are prohibited during a simulation.

3 The **power** switch has an off (down) and an on (up) position. Turn the power on by clicking the **power** switch from off to on. (This may require switching off, then switching on again.)

4 Ensure that the **running speed** switch is in the normal position.

5 If it is not already visible, double-click on the Scope block.

6 The **start** switch is a toggle. Click the **start** switch from up to down or from down to up to start the carousel. Observe the speed in the Scope block. The ride runs for Duration seconds but can be stopped prematurely by changing the **running speed** switch to overspeed or by clicking the **emergency** switch.

7 The **emergency** switch is also a toggle. Click the **emergency** switch from up to down or from down to up to bring the carousel to an emergency stop.

8 Choose **Stop** from the graphics editor **Simulation** menu to stop the simulation (and unfreeze the graphics editor). The simulation must be stopped to be able to close the model.

9 Choose **Close** from the **File** menu of the carousel model solution window.

Run the Carousel Demo

Overview

Audio and video have been added to the carousel demo. The demo also uses many advanced features that are left to your experimentation and investigation.

On UNIX systems, the audio portion of the demo is disabled.

Carousel Operator's Console

As in the carousel solution example, the switches in the Simulink model are used to act as a carousel operator's console. The switches allow you to:

- Turn the carousel power on and off
- Start the ride
- Stop the ride in case of emergency
- Specify whether the speed is in the normal or overspeed range

Procedure

Follow these steps to run the demo:

1. Double-click on the Examples block in the Stateflow model window. Double-click on the Carousel and then Multi-Media Carousel block.

2. Choose **Start** from the graphics editor **Simulation** menu.

   The first time you run the simulation, and any time you have modified the Stateflow diagram, choosing **Start** will parse the machine, generate simulation code and generate the simulation target. If the target build takes

place, informational messages are displayed in popup windows and in the MATLAB command window.

In any case, the graphics editor is 'ICED' and modifications to the Stateflow diagram are prohibited during a simulation.

3. Turn the power on by clicking the **power** switch from off to on.

4. Ensure that the **running speed** switch is in the normal position.

5. Click the **start** switch on to start the carousel. The ride runs for `Duration` seconds but can be stopped prematurely by changing the **running speed** switch to overspeed or by clicking the **emergency** switch.

   If you have sound support (sound board, driver, and speakers), you should hear the music. You should also see a representation of the carousel platform rotating in XY Graph figure window. The Stateflow diagram has animation enabled, so you can see the state transitions occur as the simulation executes.

6. Click the **emergency** switch to bring the carousel to an emergency stop.

7. Choose **Stop** from the graphics editor **Simulation** menu to stop the simulation (and unfreeze the graphics editor). The simulation must be stopped to be able to close the model.

8. Choose **Close** from the **File** menu of the carousel demo model window.

9

Generating Code

Overview 9-2

Using the Notation Efficiently 9-7

Parsing 9-11

Adding and Specifying a Stand-Alone Target 9-16

Customizing the Target 9-20

Compiling Code/Building Targets 9-26

Typical Code Generation Related Error Messages . . . 9-30

Overview

Overview

Knowledge of the semantics is important to make sound Stateflow diagram design decisions and for code generation. The default semantics provided with the product are described in Chapter 7, “Semantics.”

The Stateflow notation supports many different ways of representing desired system behavior. The representation you choose directly affects the efficiency of the generated code. See “Using the Notation Efficiently” on page 9-7 for more information.

The Stateflow simulation code generator generates code for simulation targets. Stateflow Coder generates code for stand-alone or RTW targets. This chapter discusses both types of code generation.

Stateflow Coder

Stateflow Coder is a separately available product for use with Stateflow to generate code for RTW and Stateflow stand-alone (nonsimulation/non-RTW) targets. You need a Stateflow Coder license to be able to generate RTW and Stateflow stand-alone target code. (Using Stateflow you can generate simulation target code only.) Stateflow Coder has these additional features:

- Generated code is immediately very readable including fully commented sections. Object names in the Stateflow diagram are preserved in the generated code. In addition, the first 24 characters of all symbols are guaranteed to be unique ensuring portability to many embedded system’s target cross-compilers.
- All data objects are typed, which ensures more efficient generated code.
- Implementation of flow diagram notation (especially those including transition actions) is highly optimized.
- Function inlining is used for optimizing event broadcasts.
- Stateflow Coder takes advantage of any directed event broadcasting and thus ensures minimal overhead during that event broadcast.

Stateflow Dynamic Checker

Stateflow Dynamic Checker is currently available if you have a Stateflow license.

Stateflow Dynamic Checker is useful for debugging. During a simulation the Dynamic Checker evaluates the coverage of the simulation (i.e., the percentage of unprocessed transitions, states that are visited). In addition you can choose to detect these conditions: state inconsistencies, data range violations, transition conflicts, and cyclic behavior.

Generally speaking, the Dynamic Checker is used for simulation, and options should be disabled for RTW and stand-alone targets. The overhead incurred from the added instrumented code is undesirable in these targets.

Enabling most of the dynamic checking options adds instrumentation code to the target.

See “Enable Dynamic Checker Options” on page 9-24 and “Dynamic Checker Options” on page 10-10 for more information.

Compile/Build Options

Using Stateflow and Stateflow Coder you can easily define, build, and manage a variety of targets using the Explorer and the **Target configuration** dialog boxes. These options exist for compiling code and building targets:

- `sfun` simulation target generation

  Stateflow generates the simulation target. Stateflow and Simulink are required.

  There is a single `sfun` target provided by default with Stateflow. The target code in this case is an S-function (MEX-file). See the *Using Simulink* document for information on S-functions. See “Customizing the Target” on page 9-20 and “Compiling/Building a Simulation Target” on page 9-27 for more information.

- RTW target generation

  RTW generates the target including code generated from Stateflow Coder. Stateflow, Stateflow Coder, Simulink, and RTW are required. Building a RTW target is an example of a prototype target. See “Customizing the

Target” on page 9-20 and “Compiling/Building Targets with RTW” on page 9-27 for more information.

- Stand-alone (nonsimulation/non-RTW) target generation

  Stateflow Coder generates the target. Stateflow and Stateflow Coder are required. Using Stateflow Coder you can create multiple targets using specific compilers for prototyping or production use on particular hardware platforms. See “Adding and Specifying a Stand-Alone Target” on page 9-16, “Customizing the Target” on page 9-20, and “Compiling/Building Stand-Alone Targets” on page 9-28 for more information.

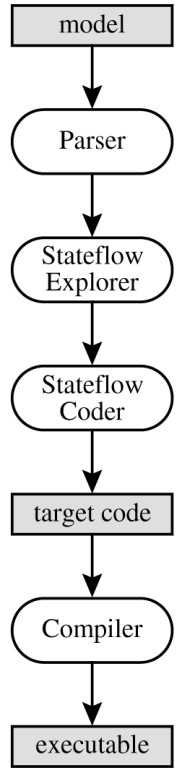
Typical Tasks to Generate Code

These are some typical code generation tasks you might want to accomplish:

- Create a Stateflow diagram using the notation efficiently (knowing the semantics) for code generation
- Parse the machine
- Add and specify a stand-alone target
- Customize the target
- Compile code and build a target

Typical Code Generation Sequence

This diagram shows a typical progression from model creation to generated executable assuming you are generating code for a stand-alone target.



Parse the model; invoke via **Tools** menu or **Target** dialog box.

Add a target and customize the commands that generate target code in the **Target Configuration** dialog box; invoke via **Tools** or any shortcut menu.

Generate code; invoke explicitly via the **Target Configuration** dialog box or implicitly via **Start** menu item on the **Simulation** menu.

Generate executable; explicitly invoke the compiler via the **Target Configuration** dialog box or implicitly via the **Start** menu item on the **Simulation** menu.

Coder File and Coder Function

The coder file (.cdr) is an ASCII template file that is used to generate target language code. The .cdr file implements a defined set of semantics for a particular notation.

Stateflow provides the coder file called `sfc.cdr` to generate C language code. Stateflow generates two target code files (`sfc.m` and `sfc.p`) that have the same name as the target.

Using the Notation Efficiently

Where to Find Information on Creating a Stateflow Diagram

See Chapter 2, “Using the Graphics Editor,” Chapter 6, “Notations,” and Chapter 7, “Semantics” for information on creating Stateflow diagrams. See Chapter 4, “Defining Stateflow Interfaces” for information on defining events and data objects.

Designing for Efficiency

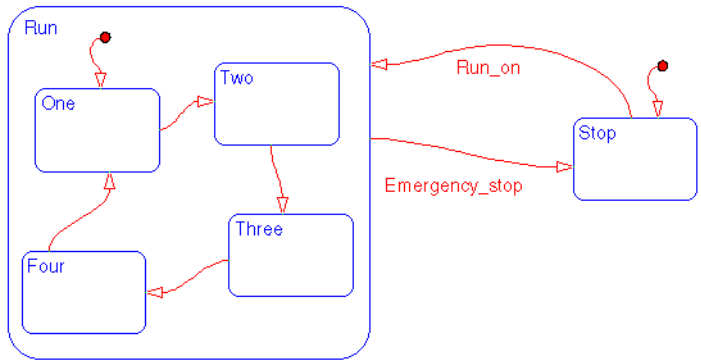
The Stateflow notation supports many different ways of representing desired system behavior. The representation you choose directly affects the efficiency of the generated code. You can use the notation to optimize for these code attributes:

- Speed
- Code size
- Memory use

Optimizing for Speed

If execution speed is the most important efficiency criterion for the generated code, take advantage of the top-down processing of events. Designing the Stateflow diagram so that execution of speed-critical actions occurs at the highest hierarchy level possible will produce speed-efficient code. Taking advantage of the ability to specify an event's parent (within the hierarchy) provides speed optimizations. You can make sure that an event's broadcast scope is as specific as possible to avoid unnecessary event broadcasting. See “Events” on page 3-5 and “Event Inputs from Simulink” on page 4-16 for more information.You can also use the send action language function to further qualify the receiver of an event broadcast (directed event broadcasting). See “Directed Event Broadcasting” on page 6-58 for more information.

This example Stateflow diagram shows the use of top-down event processing to optimize the event-handling speed.



If the emergency stop button is pushed, the event `Emergency_stop` occurs and the system should transition to the `Stop` state as quickly as possible. By drawing the `Emergency_stop` transition from the `Run` superstate border to the `Stop` state, the transition is evaluated at the highest level in the hierarchy. Using this notation, if any of `Run` substates are active and the `Emergency_stop` event occurs, the system transitions to the `Stop` state. The alternative to this notation is to have individual transitions (originating from a lower level in the hierarchy) from each of `Run`'s substates. The event handling of the alternative is less speed efficient.

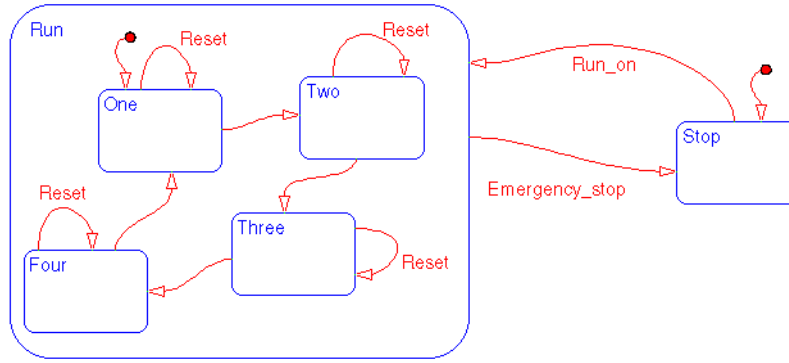
See “Example: Directed Event Broadcast Using send” on page 6-59 and “Example: Directed Event Broadcast Using Qualified Event Names” on page 6-60 for examples using the send action language function.

Optimizing for Code Size

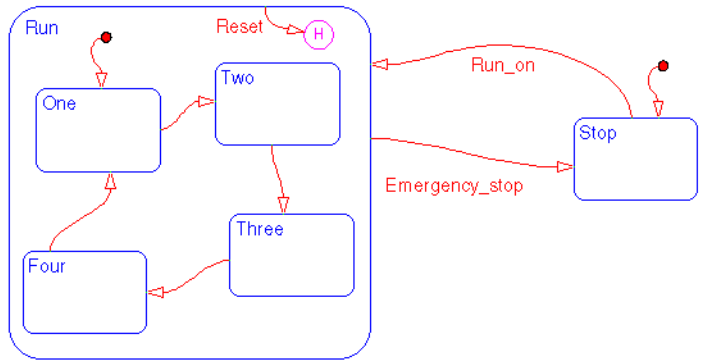
Simplifying Stateflow diagrams results in less generated code. The fewer transitions, the simpler the Stateflow diagram is. Use of hierarchy, history junctions, and inner transitions reduces the number of transitions.

In this example, if event `Reset` occurs and any substate of `Run` is active, you want to exit and re-enter that same substate. For example, when the substate

is re-entered, its entry action is executed. This example uses many transitions to represent the logic.



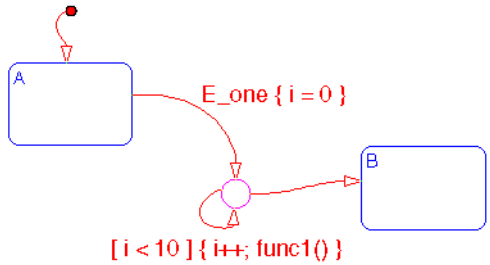
This is an equivalent, simpler Stateflow diagram that uses an inner transition to a history junction to optimize for code size.



Optimizing for Memory Use

The use of flow diagram notation in a Stateflow diagram can produce efficient code optimized for memory use. In some cases, specifying actions using flow diagram notation is a closer representation of the system's logic and avoids the use of unnecessary states. Reducing the number of states optimizes for memory use.

This example shows the use of flow diagram notation to represent a `for` loop.



The transition from state `A` to state `B` is divided into segments. Two of the segments are labeled with condition actions `{i = 0}` and `{i++; func1()}`. Using the Connective junction and condition action notation in this manner is a closer representation of a `for` loop than could be otherwise represented using states. The elimination of unnecessary states also optimizes the use of memory.

For data that does not need to be persistent, use of **Temporary** data objects provides a memory-use optimization. Defining `i` to be a **Temporary** data object (parented by the chart) also improves memory use. See “Data Scope” on page 3-27 for more information on **Temporary** data objects.

Parsing

Parser

The parser evaluates the graphical and nongraphical objects in each Stateflow machine against the supported Stateflow notation and the action language syntax.

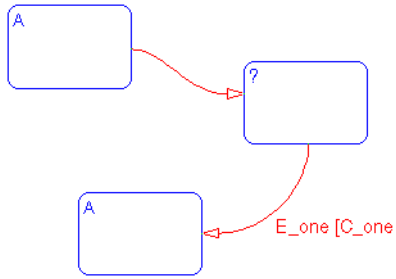
Parse the Machine or the Stateflow Diagram

Explicitly parse each Stateflow diagram in the machine by choosing **Parse** from the graphics editor **Tools** menu. Explicitly parse the current Stateflow diagram by choosing **Parse Diagram** from the graphics editor **Tools** menu. The machine is implicitly parsed before code generation by:

- Choosing **Start** from the graphics editor **Simulation** menu.
- Choosing **Start** from the Simulink model **Simulation** menu.
- Choosing **Open Simulation Target** from the graphics editor **Tools** menu to display the **Target Configuration sfun** properties dialog box. Click on the **Generate Code Only** or **Build Stateflow Target** buttons in the **Target Configuration sfun** dialog box.
- Choosing **Explore** from the graphics editor **Tools** menu to display the Explorer. Double-click on a stand-alone target to display its **Target Configuration** properties dialog box. Click on the **Generate Code Only** or **Build Stateflow Target** buttons in the **Target Configuration** dialog box.
- Choosing **Parameters...** from a **Simulation** menu (either the Simulink model or the graphics editor menu). Click on the RTW tab. Click on the **Build** button in the RTW tab of the **Simulation parameters** dialog box.

In all cases, a popup information window is displayed when the parsing is complete. If the parsing is unsuccessful, one error at a time is displayed (in red) in the informational window. The Stateflow diagram automatically selects and pans to the object containing the parse error. Double-click on the error in the information window to bring the Stateflow diagram to the forefront, zoom (fit to view), and select the object containing the parse error. Press the space bar to zoom back out. Fix the error and reparse the Stateflow diagram. Informational messages are also displayed in the MATLAB command window.

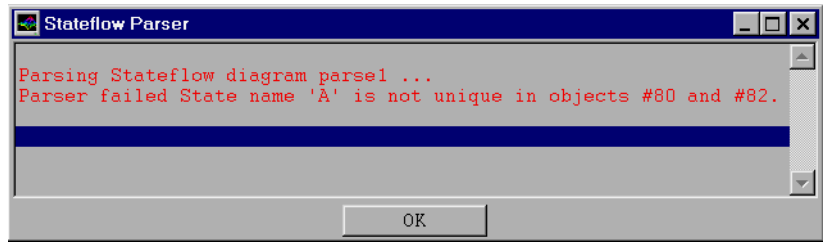
These steps describe parsing, assuming this Stateflow diagram.



1 Parse the Stateflow diagram.

Choose **Parse Diagram** from the graphics editor **Tools** menu to parse the Stateflow diagram. State A in the upper left-hand corner is selected and this

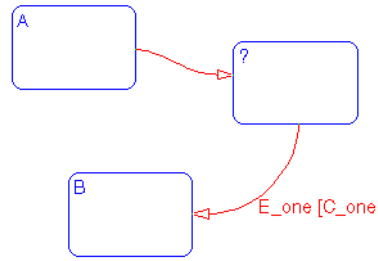
message is displayed in the popup window and the MATLAB command window.



2 Fix the parse error.

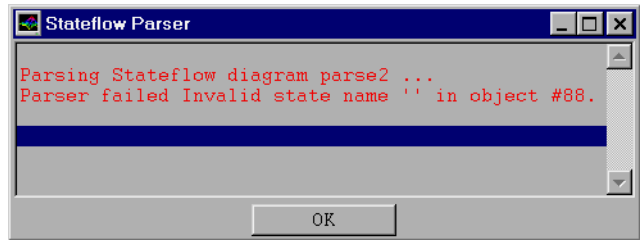
In this example, there are two states with the name `A`. Edit the Stateflow diagram and label the duplicate state with the text `B`.

The Stateflow diagram should look similar to this.



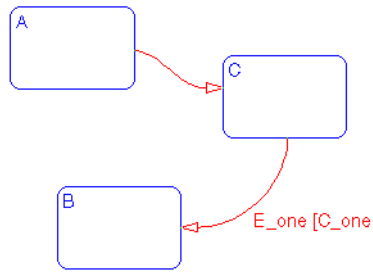
3 Reparse.

Choose **Parse Diagram** from the graphics editor **Tools** menu. This message is displayed in the popup and the MATLAB command window.



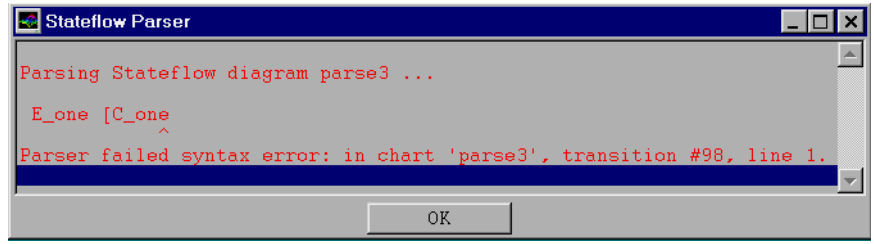
4 Fix the parse error.

In this example, the state with the question mark needs to be labeled with at least a state name. Edit the Stateflow diagram and label the state with the text `C`. The Stateflow diagram should look similar to this.



5 Reparse.

Choose **Parse Diagram** from the graphics editor **Tools** menu. This message is displayed in the popup window and the MATLAB command window.

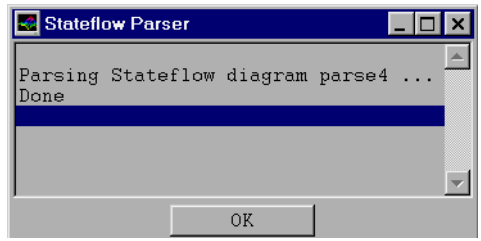

Stateflow Parser
Parsing Stateflow diagram parse3 ...
E_one [C_one
Parser failed syntax error: in chart 'parse3', transition #98, line 1.
OK

6 Fix the parse error.

In this example, the transition label contains a syntax error. The closing bracket of the condition is missing. Edit the Stateflow diagram and add the closing bracket so that the label is `E_one [C_one]`.

7 Reparse.

Choose **Parse Diagram** from the graphics editor **Tools** menu. This message is displayed in the popup window and the MATLAB command window.



The Stateflow diagram has no parse errors.

Adding and Specifying a Stand-Alone Target

Stand-Alone Targets

The Stateflow Coder product is required to generate code for stand-alone and RTW targets.

Invoke the Explorer

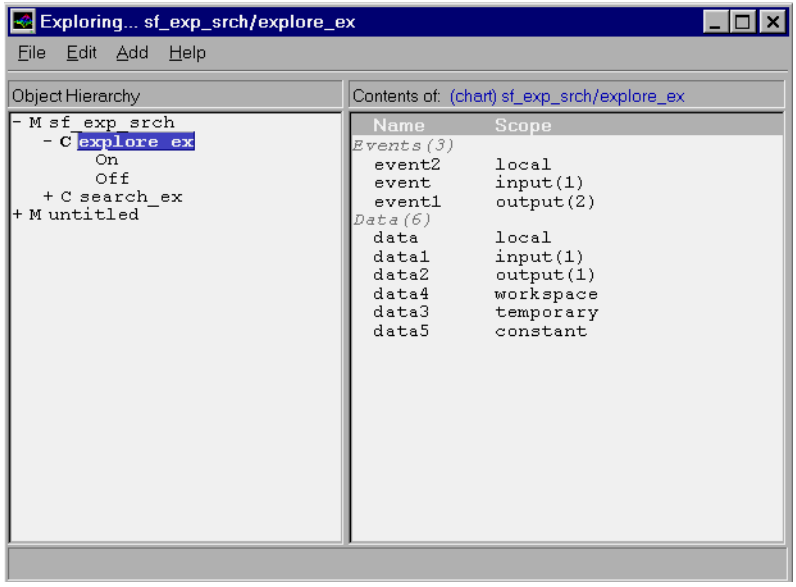
This section shows how to add a stand-alone target using the Explorer. You can also add a stand-alone target by choosing **Target** from the graphics editor **Add** menu.

To invoke the Explorer:

1 Open the Stateflow diagram.

2 Choose **Explore** from the graphics editor **Tools** menu.

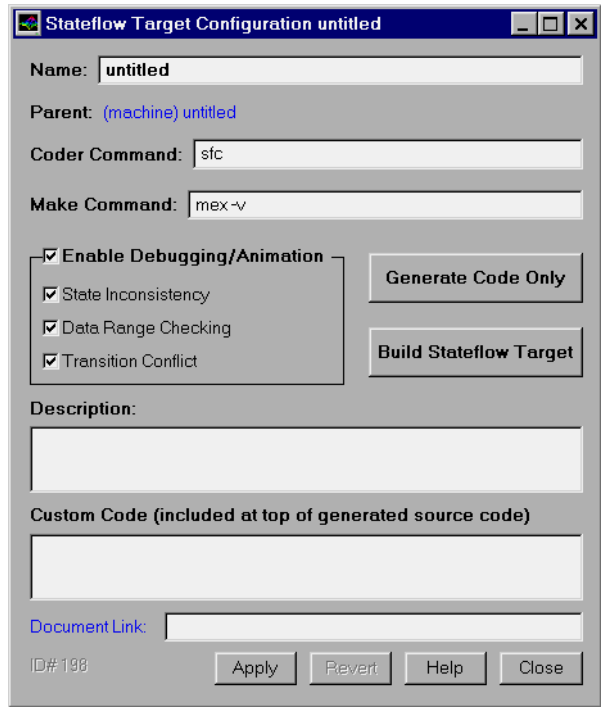
The Explorer is displayed.



Add a Stand-Alone Target

These steps show how to add a stand-alone target.

1 In the Explorer window, select a machine as the target's parent. Choose **New** from the Explorer **File** menu. The **Stateflow Target Configuration** dialog box is displayed.



2 Enter the target name in the **Name** field.

3 Specify the **Coder Command**.

The **Coder Command** field defines how the code is to be generated. The code command consists of the name of the coder function (without the file extension) followed by any flags and the target name. The default coder function

provided is `sfc`. See “Coder Command Customization” on page 9-23 for more information.

Click on the **Generate Code Only** button to parse the machine implicitly and generate target code. Informational messages and any error messages are displayed in a pop-up window and also in the MATLAB command window.

4 Specify the **Make Command**.

The **Make Command** field defines how the executable is to be generated and varies depending on your environment. See “Coder Command Customization” on page 9-23 for more information.

Click on the **Build Stateflow Target** button to generate an executable file. This operation implicitly parses the machine, generates code, and generates an executable image. You can build one target at a time. Informational and any error messages are displayed in a pop-up window and also in the MATLAB command window.

5 Disable debugging/animation.

Generally speaking, debugging/animation should be disabled for stand-alone targets. The debugger does not interact with stand-alone

targets and the overhead incurred from the added instrumented code is undesirable in stand-alone targets.

Ensure that the **Enable Debugging/Animation** checkbox is disabled. See “Enable Debugging/Animation” on page 9-24 and Chapter 10 “Debugging” for more information.

6 Enable Dynamic Checker options.

Generally speaking, the Dynamic Checker options should be disabled for stand-alone targets. The overhead incurred from the added instrumented code is undesirable in stand-alone targets.

Ensure that the appropriate check boxes are disabled. See “Enable Dynamic Checker Options” on page 9-24 and Chapter 10 “Debugging” for more information.

7 Specify any custom code.

Enter any customized code that you want to be prepended to the generated target code in this field. See “Add Customized Code” on page 9-25 for more information.

8 Specify a document link.

See “Specify a Document Link” on page 9-25 for more information.

9 Save and apply the target information.

Click the **Apply** button to save and apply the target information you have specified. Click on the **Close** button to save the changes and close the dialog box.

Customizing the Target

Overview

You can customize the default `sfun` simulation target, any stand-alone target you have added as well as the `rtw` target (if you are using Real-Time Workshop). You can customize the `sfun` target using the **Target Configuration sfun** dialog box. You can customize the `rtw` target using the **Target Configuration rtw** dialog box. You can customize any stand-alone target by using its **Target Configuration** dialog box.

Potential Customizations

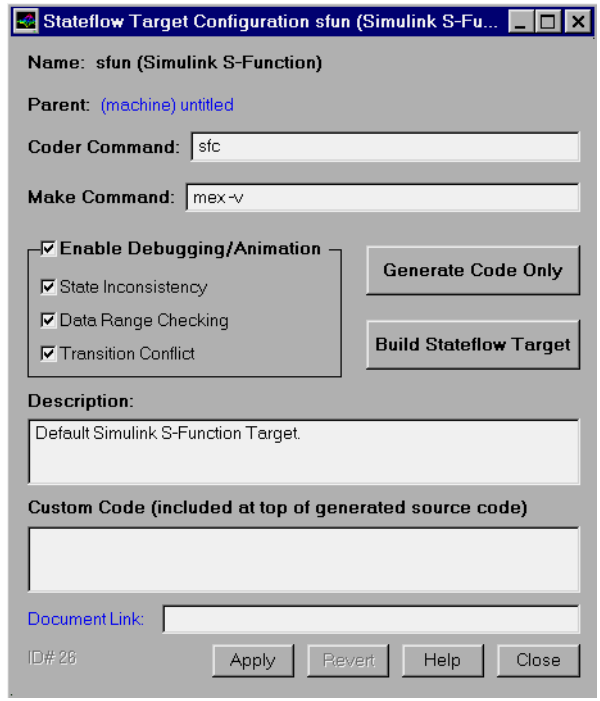
These are common customizations you can make to the target:

- Modify the **Coder Command** field
- Modify the **Make Command** field (stand-alone targets only)
- Enable Debugging/Animation (simulation targets only)
- Enable Dynamic Checker options (simulation targets only)
- Add any customized code that you want to be prepended to the generated target code in the Custom Code field.
- Specify a document link.

Once you have made the customizations, click on the **Apply** button to save changes. Click on the **Revert** button to remove changes and reset the fields with the previous values. Click on the **Close** button to apply changes and close the window.

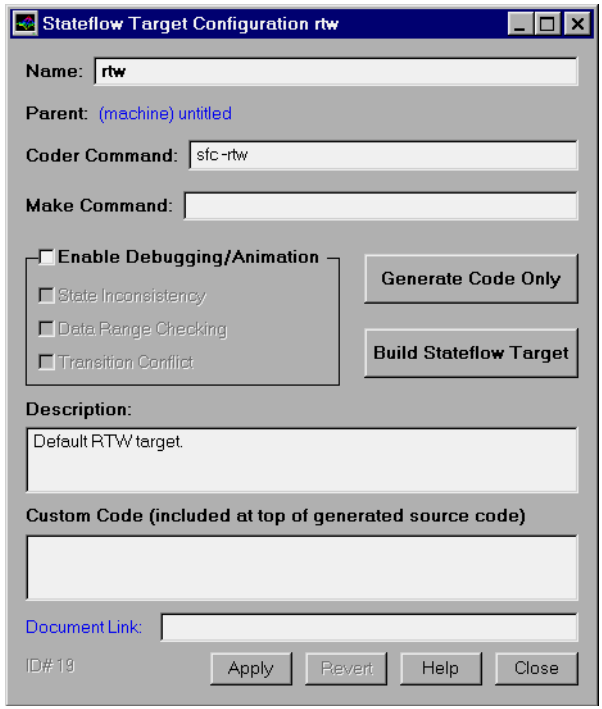
Displaying the Target Configuration sfun Dialog Box

Choose **Open Simulation Target** from the graphics editor **Tools** menu to display the **Target Configuration sfun** dialog box.



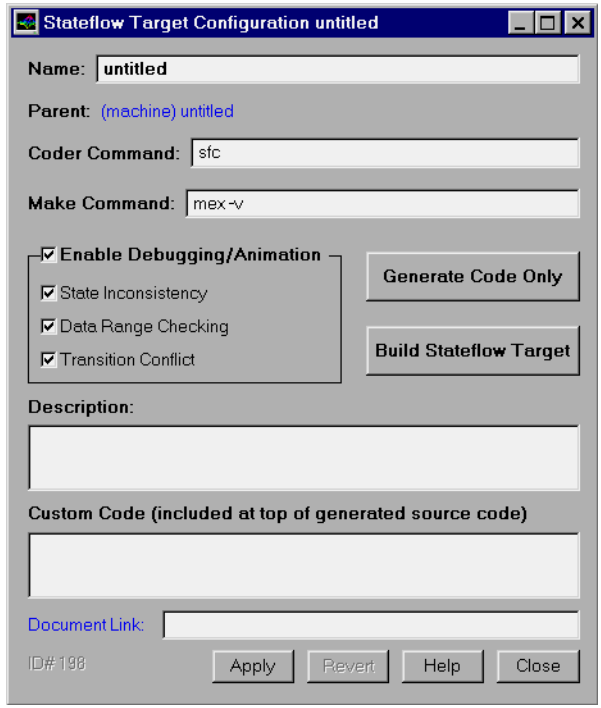
Displaying the Target Configuration rtw Dialog Box

Choose **Open RTW Target** from the graphics editor **Tools** menu to display the **Target Configuration rtw** dialog box.



Displaying the Target Configuration Dialog Box

Choose **Explore** from the graphics editor **Tools** menu to display the Explorer. Double-click on the target you want to customize to display the Target Configuration dialog box.



Coder Command Customization

You can add `+ bitops` to the code generation command in the **Coder Command** field to enable C-like bit operations. Removing the `+ bitops` disables C-like bit operations. C-like bit operations are disabled by default. See “Action Language” on page 6-42 for information on these operations.

Make Command Customization

By default, the command to generate an executable file for the `sfun` target is `mex -v`. At the MATLAB prompt, type `help mex`.

for information on the MEX command and flags. Also see the *MATLAB Application Program Interface Guide* for more information. For the sfun target, you can customize the MEX command flags but do not change the command itself.

By default, the command to generate an executable file for the rtw target is sfc –rtw. The **Make Command** field for the rtw target is read-only.

You can fully customize the **Make Command** field for stand-alone targets including specifying inclusion of external code sources.

A make utility may be included with your operating system (such as on UNIX systems), supplied with your compiler package (such as the WATCOM wmake utility), or part of a separate package (like GNU gmake utility). While there are differences between various make utilities, their operation is similar. See the *Real-Time Workshop User's Guide* for more information.

Enable Debugging/Animation

If you want debugging/animation options to be available during simulation, you need to include them by clicking in the **Enable Debugging/Animation** checkbox. Enabling debugging/animation options adds instrumentation code to the target.

Generally speaking, **Debugging/Animation** should be disabled for stand-alone targets. The Debugger does not interact with stand-alone targets and the overhead incurred from the added instrumented code is undesirable in stand-alone targets.

Debugging/Animation should always be disabled for rtw targets. The Debugger does not interact with rtw targets.

Enable Dynamic Checker Options

Click on the checkboxes in the **Target Configuration** dialog box to include the necessary code for these dynamic checks:

- State inconsistencies

  State inconsistencies can occur as a result of event processing. For example, at least one, but only one substate of a superstate with exclusive (OR)

decomposition should be active. Another example is that all substates of a superstate with parallel (AND) decomposition should be active.

- Data range violations

  Data range checking verifies that a data object value does not assume a value less than the minimum nor greater than the maximum values specified in the object's definition.

- Transition conflicts

  Transition conflicts exist if more than one transition from the same source are equivalently valid at any one time.

Generally speaking, the Dynamic Checker options should be disabled for stand-alone targets. The overhead incurred from the added instrumented code is undesirable in stand-alone targets.

The Dynamic Checker options should always be disabled for `rtw` targets. The Debugger does not interact with `rtw` targets.

The Debugger displays results of these dynamic checks as well as cycle detection at runtime. Cycle detection does not require any instrumented code and is thus controlled solely by the Debugger. The cycle detection algorithm can detect certain classes of runtime infinite loops (recursions that repeat forever). These loops are caused by recursive event broadcasts.

See Chapter 10 “Debugging” and specifically “Dynamic Checker Options” on page 10-10 for information on how to use the Debugger in conjunction with these options and also the cycle detection option.

Add Customized Code

You can add customized code that is prepended to the generated target code in the **Custom Code** field. For example you might want to specify inclusion of custom header files and declarations of user-written functions. Any code specified here is considered external to the Stateflow diagram, the machine, and the Simulink model.

Specify a Document Link

For convenience, enter a web URL address or a general MATLAB command. Examples are: `www.mathworks.com`, `mailto:email_address`, `edit/spec/data/speed.txt`. The contents of this field are not represented in the generated code.

Compiling Code/Building Targets

Compile/Build Options

These options exist for compiling code and building targets:

- `sfun` simulation target generation

  Stateflow generates the simulation target. Stateflow and Simulink are required.

  There is a single `sfun` target provided by default with Stateflow. The target code in this case is an S-function (MEX-file). See the *Using Simulink* document for information on S-functions. See "Customizing the Target" on page 9-20 and "Compiling/Building a Simulation Target" on page 9-27 for more information.

- RTW target generation

  RTW generates the target including code generated from Stateflow Coder. Stateflow, Stateflow Coder, Simulink, and RTW are required. Building a RTW target is an example of a prototype target. See "Customizing the Target" on page 9-20 and "Compiling/Building Targets with RTW" on page 9-27 for more information.

- Stand-alone (nonsimulation/non-RTW) target generation

  Stateflow Coder generates the target. Stateflow and Stateflow Coder are required. Using Stateflow Coder you can create multiple targets using specific compilers for prototyping or production use on particular hardware platforms. See "Adding and Specifying a Stand-Alone Target" on page 9-16, "Customizing the Target" on page 9-20, and "Compiling/Building Stand-Alone Targets" on page 9-28 for more information.

Compiling/Building a Simulation Target

These steps show how to build a simulation target assuming you have already specified any customizations (see “Customizing the Target” on page 9-20).

1. Choose **Open Simulation Target** from the graphics editor **Tools** menu to display the **Target Configuration sfun** dialog box.
2. Click on the **Generate Code** button in the **Target Configuration sfun** dialog box. This operation implicitly parses the machine and generates the S-function code.
3. Click on the **Build Stateflow Target** button in the **Target Configuration sfun** dialog box to build the simulation target. This operation implicitly parses the machine, generates code, and generates an executable image. You can build one target at a time.

Alternatively you can build the `sfun` target through either of these options:

- Choosing **Start** from the graphics editor **Simulation** menu.
- Choosing **Start** from the Simulink model **Simulation** menu.
- Choose **Debug** from the graphics editor **Tools** menu, choose debugging options, and click on the **Go** button.

Each of these options implicitly parses the machine, generates code, and generates an executable image.

Informational and any error messages related to the S-function code generation are displayed in a pop-up window and also in the MATLAB command window.

See the *Application Program Interface Guide* for information on proper installation of the C compiler for integration into the MATLAB environment.

Compiling/Building Targets with RTW

RTW performs the target generation of a Simulink model that contains a Stateflow machine. When an RTW build is initiated, RTW automatically detects the presence of a Stateflow machine. RTW creates a Stateflow target titled `rtw` on Stateflow's behalf.

These steps show how to build a target with RTW.

1. Choose **Parameters...** from a **Simulation** menu (either the Simulink model or the graphics editor menu).
2. Click on the **RTW** tab.
3. Specify any RTW specific options in the **RTW** tab. (See the *Real-Time Workshop User's Guide* for detailed information on RTW.)
4. Click on the **Build** button in the **RTW** tab of the **Simulation parameters** dialog box.

Stateflow Coder must be present to build a target with RTW.

Calls to MATLAB functions in the action language should be avoided if you plan to build an RTW target that includes code from Stateflow Coder.

Compiling/Building Stand-Alone Targets

These steps show how to build a stand-alone Stateflow target assuming you have already specified any customizations (see "Customizing the Target" on page 9-20).

1. Choose **Explore** from the graphics editor **Tools** menu to invoke the Explorer.
2. Click on the machine of interest in the **Object Hierarchy** list (left-hand side of the Explorer).
3. Double-click on the target name in the Explorer window to display the **Target Configuration** property dialog box.
4. Click on the **Generate Code** button in the **Target Configuration** dialog box to generate the code. This operation implicitly parses the machine and generates the code.
5. Click on the **Build Stateflow Target** button in the **Target Configuration** dialog box to build the stand-alone target. This operation implicitly parses the machine, generates code, and generates an executable image. You can build one target at a time.

Informational messages and any error messages are displayed in a popup window and also in the MATLAB command window.

Stateflow Coder must be present to build stand-alone targets.

Typical Code Generation Related Error Messages

Overview

Code generation related error messages can originate from the parser, the simulation code generator, Stateflow Coder, or from the external C compiler. If you are explicitly parsing, generating code, or building a target it is apparent which operation generated the error.

It is less clear which operation generated the error if you are implicitly performing these operations through one of these methods:

- Choosing **Start** from the graphics editor **Simulation** menu.
- Choosing **Start** from the Simulink model **Simulation** menu.
- Choosing **Debug** from the graphics editor **Tools** menu, choosing debugging options, and clicking on the **Go** button.

A common error message when a simulation is started via one of those options is:

```
"Error using => toolbox/stateflow/stateflow/private/
(intfcn_method). Stateflow S-Function build failed."
```

If you receive this message, you may have parser, code generation, or target generation errors.

Errors are displayed (in red) in an informational dialog box and also in the MATLAB command window. Note that whenever you see a line in the error message dialog box , you can double-click on that line to zoom in (in the Stateflow diagram) on the object.

Typical Parsing Error Messages

Parse errors are usually caused by misrepresentations in the notation (essentially syntax errors). A common error in the notation is to use a backward slash (\) instead of a forward slash (/) to separate the transition action from the condition action. This will generate a general parse error.

Typical parse error messages include:

- "`Invalid state name xxx`" or "`Invalid event name yyy`" or "`Invalid data name zzz`"

  These parse errors are generated because non-alphanumeric characters (such as spaces, # , . , ?, etc.) were used in state, data, or event names. The only non-alphanumeric character allowed in state, data, or event names is an underscore.

- "`State name xxx is not unique in objects #yyy and #zzz`"

  This parse error is generated because there are two or more states at the same hierarchy level with the same name.

- "`Invalid transition out of AND state xxx (#yy)`"

  This parse error is generated because there is a transition originating from an AND (parallel) state. The notation does not support this kind of transition.

- "`Invalid intersection between states xxx and yyy`"

  This parse error is generated because neighboring state borders are intersecting. If the intersection is not apparent, consider the state to be a cornered rectangle instead of a rounded rectangle. Ensure that the borders of the state, if it were a cornered rectangle, are not intersecting.

- "`Junction #x is sourcing more than one unconditional transition`"

  The notation specifies that only one transition originating from a connective junction can be unconditional (that is without a [condition] as part of its label).

- "`Multiple history junctions in the same state #xxx`"

  The notation specifies there can be only one history junction within a particular state.

Code Generation Error Messages

Typical code generation error messages include:

- "Failed to create file: modelName_sfun.c"

  This code generation error message is generated if you do not have write permission for the current directory. Either change directory to a writeable one or change the permissions on the current directory to fix this error.

- "Another unconditional transition of higher priority shadows transition # xx"

  This code generation error message is generated if there is more than one unconditional inner, default, or outer transition originating from the same source.

- "Default transition cannot end on a state that is not a substate of the originating state."

  This code generation error is generated when a transition path starting from a default transition segment in one state completes at a destination state that is not a substate of the original state. This notation is not supported because it causes infinite recursions in the generated code. The solution is to change the design so that the above rule is not violated.

- "Input data xxx on left hand side of an expression in yyy"

  This code generation error is generated if the **Input from Simulink** data object indicated is appearing on the left hand side of the expression specified. By definition, the value of the data object is assigned in Simulink; Stateflow cannot change the value.

Compilation Error Messages

If compilation errors indicate the existence of undeclared identifiers, verify that variable expressions in state, condition, and transition actions are defined.

Consider, for example, an action language expression such as `a=b+c`. In addition to entering this expression in the Stateflow diagram, you must create data objects for `a`, `b`, and `c` using the Explorer. If the data objects are not defined, the parser assumes that these unknown variables are defined in the **Custom code** portion of the target (which is included at the beginning of the generated code). This is why the error messages are encountered at compile time and not at code generation time.

10

Debugging

Overview 10-2

Stateflow Debugger User Interface 10-6

Debugging Runtime Errors 10-12

Debugging State Inconsistencies 10-16

Debugging Conflicting Transitions 10-18

Debugging Data Range Violations 10-20

Debugging Cyclic Behavior 10-21

Overview

Debugging

Use the Stateflow Debugger to debug and animate the Stateflow diagrams in a particular machine.

It is a good idea to include debugging options in preliminary simulation target builds to ensure that the model is behaving as you expect, to evaluate code coverage, and to perform dynamic checking.

When you save the Stateflow diagram, all of the debugger settings (including breakpoints) are saved.

Generally speaking, debugging options should be disabled for RTW and standalone targets. The Debugger does not interact with RTW or standalone targets and the overhead incurred from the added instrumented code is undesirable.

Stateflow Dynamic Checker

Stateflow Dynamic Checker is currently available if you have a Stateflow license.

Stateflow Dynamic Checker is useful for debugging. During a simulation, the Dynamic Checker evaluates the coverage of the simulation (i.e., the percentage of unprocessed transitions, states that are visited) In addition you can choose to detect these conditions: state inconsistencies, data range violations, transition conflicts, and cyclic behavior.

Generally speaking, the Dynamic Checker is used for simulation, and options should be disabled for RTW and standalone targets. The overhead incurred from the added instrumented code is undesirable in these targets.

Enabling most of the dynamic checking options adds instrumentation code to the target.

See “Enable Dynamic Checker Options” on page 9-24 and “Dynamic Checker Options” on page 10-10 for more information.

Typical Debugging Tasks

These are some typical debugging tasks you might want to accomplish:

- Animate Stateflow diagrams, set breakpoints, and debug runtime errors
- If you are licensed for the Stateflow Dynamic Checker you can detect these conditions:
  - Evaluate code coverage
  - State inconsistencies
  - Conflicting transitions
  - Data range violations
  - Cyclic behavior

Including Debugging in the Target Build

These debugging options require supporting code additions to the target code generated:

- State inconsistency
- Transition conflict
- Data range violations

To include the supporting code for these debugging options:

- Enable debugging and animation in the appropriate **Target Configuration** properties dialog box
- Enable the dynamic checking options of interest in the appropriate **Target Configuration** properties dialog box

You must rebuild the target for any changes made to the settings in the **Target Configuration** properties dialog box to take effect. See “Stand-Alone Targets” on page 9-16, and “Customizing the Target” on page 9-20 for more information.

Breakpoints

A breakpoint indicates where and when the Debugger should break execution of a Stateflow diagram. The Debugger supports global and local breakpoints. Global breakpoints halt execution on any occurrence of the specific type of

breakpoint. Local breakpoints halt execution on a specific object instance. When simulation execution is halted at a breakpoint, you can:

- Examine the current status of the Stateflow diagram
- Step through the execution of the Stateflow diagram
- Specify display of one of these options at a time: the call stack, code coverage, data values, or active states

The breakpoints can be changed during runtime and are immediately enforced. When you save the Stateflow diagram, all of the debugger settings (including breakpoints) are saved so that the next time you open the model, the breakpoints remain as you left them.

Runtime Debugging

Once the target is built with the debugging code, you can then optionally enable or disable the associated runtime options in the Debugger. Enabling or disabling the options in the Debugger window affects the Debugger output display results. Enabling/disabling the options in the Debugger window does not affect the target code and does not require the target to be rebuilt.

There are also some runtime debugging options that do not require supporting code in the target. These options can be dynamically set:

- Enable/disable cycle detection in the Debugger window
- Set global breakpoints at:
  - Any chart entry
  - Any event broadcast
  - Any state entry
- Enable/disable local Debugger breakpoints at specific chart or state action execution points in these appropriate property dialog boxes:
  - Chart (see “Editing Chart (Stateflow Diagram) Properties” on page 2-19)
  - State (see “Editing State Properties” on page 2-29)
- Enable/disable local Debugger breakpoints at a specific transition (either when the transition is tested or when it is determined to be valid) in the

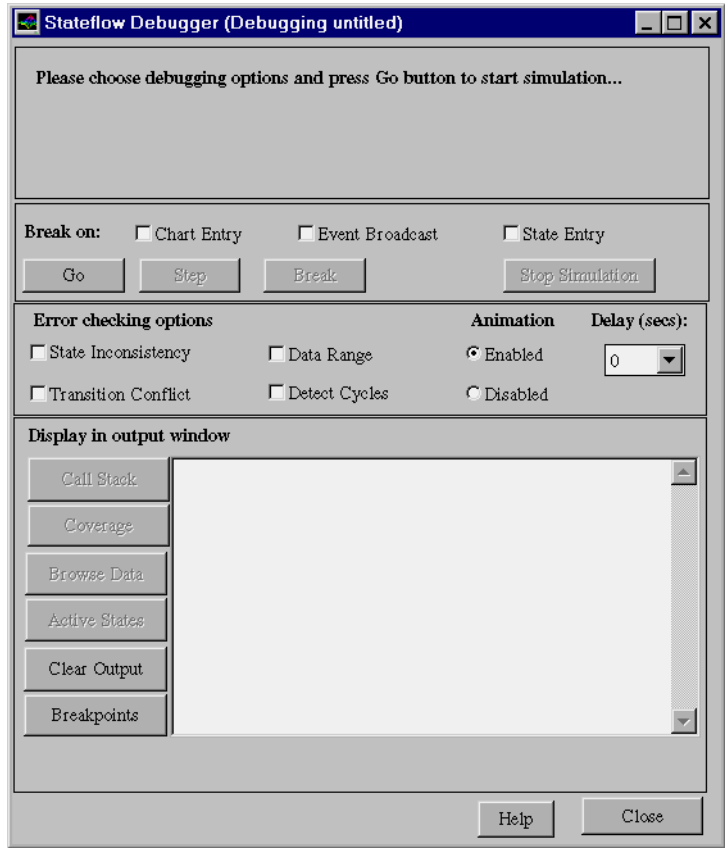
Transition property dialog box (see “Editing Transition Properties” on page 2-36)

- Enable/disable local Debugger breakpoints based on a specific event broadcast (see “Event Properties Dialog Box” on page 5-9)

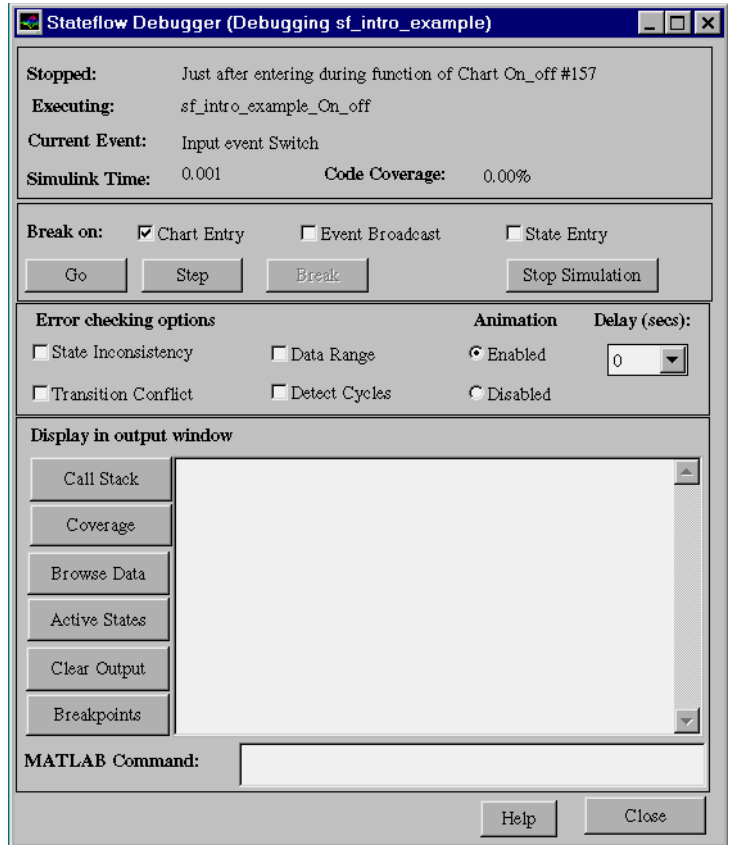
Stateflow Debugger User Interface

Debugger Main Window

This is the Debugger main window as it appears when first invoked:



This is the Debugger main window as it appears when a debug session is active:



Status Display Area

Once a debugging session is in progress, these status items are displayed in the upper portion of the Debugger window:

- The currently executing model is displayed in the **Executing** field.
- The execution point that the Debugger is halted at is displayed in the **Stopped** field. Consecutive displays of this field show each semantic step being executed.
- The event being processed is displayed in the **Current Event** field.
- The current simulation time is displayed in the **Simulation** field.
- The percentage of code that has been covered thus far in the simulation is displayed in the **Code Coverage** field.

Break On Controls

Use the **Break On** controls to specify global breakpoints. When a global breakpoint is encountered normal simulation execution stops and the Debugger takes control on any:

- Chart entry

  Click on the **Chart Entry** checkbox (check is displayed when enabled) to enable this type of breakpoint.

- Event broadcast

  Click on the **Event Broadcast** checkbox (check is displayed when enabled) to enable this type of breakpoint.

- State entry

  Click on the **State Entry** checkbox (check is displayed when enabled) to enable this type of breakpoint.

The breakpoints can be changed during runtime and are immediately enforced. When you save the Stateflow diagram, the breakpoint settings are saved.

Debugger Action Control Buttons

Use these buttons when debugging a Stateflow machine to control the Debugger's actions:

- Go

  Click on the **Go** button to have simulation execution proceed until a breakpoint (global or local) is reached. Once the **Go** button has been clicked, the Stateflow diagram is marked read-only. The appearance of the graphics editor toolbar and menus changes so that object creation is not possible. When the graphics editor is in this read-only mode, it is called 'iced'.

- Step

  Click on the **Step** button to single step through the simulation execution.

- Break

  Click on the **Break** button to insert a breakpoint at that point in the simulation execution.

- Stop Simulation

  Click on the **Stop Simulation** button to stop the simulation execution and relinquish debugging control. Once the debug session is stopped, the graphics editor toolbar and menus return to their normal appearance and operation so that object creation is again possible.

Dynamic Checker Options

The Dynamic Checker functionality is available if this product has been licensed. Enable checking for these conditions by turning on the checkbox for these options:

- State inconsistency

  Click on the **State Inconsistency** checkbox (check is displayed) to enable.

- Transition conflict

  Click on the **Conflicting Transitions** checkbox (check is displayed) to enable.

- Data range values out of specified bounds

  Data object attributes include an **Initial**, **Minimum**, and **Maximum** value. If the data object equals a value outside of this range, enabling data range checking will detect the error.

  Click on the **Data Range Check** checkbox (check is displayed) to enable.

- Cyclic behavior

  Click on the **Detect Cycles** checkbox (check is displayed) to enable.

Animation Controls

Activating animation causes visual color changes (objects are highlighted in the selection color) in the Stateflow diagram based on the simulation execution.

Activate animation by turning on the **Enabled** check box. Deactivate animation by turning on the **Disabled** check box. You can specify the animation speed from a range of 0 (fast; the default) to 1 (slow) seconds.

Display Controls

Use these buttons to control the output display:

- Call Stack

  Click on the **Call Stack** button to display a sequential list of the **Stopped** and **Current Event** status items that occur when single stepping through the simulation.

- Coverage

  Click on the **Coverage** button to display a listing of code coverage. Code coverage displays the current percentage of unprocessed transitions, states, etc. at that point in the simulation.

- Browse Data

  Click on the **Browse** button to display the current value of any defined data objects.

- Active States

  Click on the **Active States** button to bring the graphics editor in focus displaying any active states.

- Clear Output

  Click on the **Clear Output** button to remove any previously displayed information from the display area.

- Breakpoints

  Click on the **Breakpoints** button to display a list of the set breakpoints.

Once you have selected an output display button, that type of output is displayed until you choose a different display type.

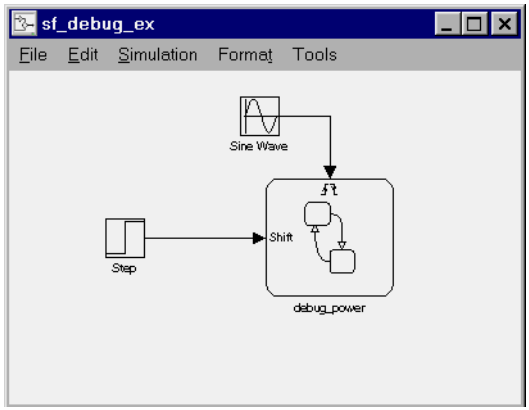
MATLAB Command Field

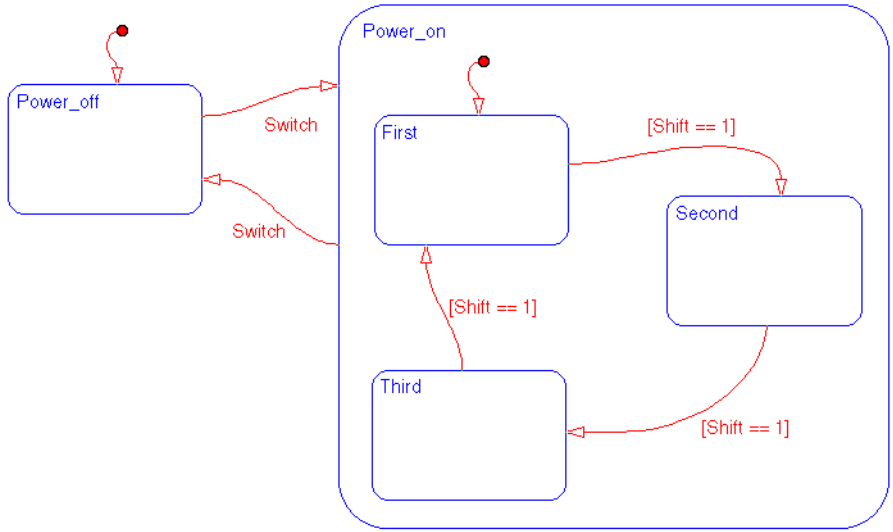
Direct access to the MATLAB command window is not possible while the Debugger is stopped at a breakpoint. If you need to enter any MATLAB commands during a debugging session, enter them into the **MATLAB Command** field and press the **Return** key.

Debugging Runtime Errors

Example Stateflow Diagram

This example Simulink model and Stateflow diagram is used to show how to debug some typical runtime errors.





The Stateflow diagram has two states at the highest level in the hierarchy, `Power_off` and `Power_on`. By default `Power_off` is active. The event `Switch` toggles the system between being in `Power_off` and `Power_on`. `Switch` is defined as an **Input from Simulink** event. `Power_on` has three substates, `First`, `Second`, and `Third`. By default, when `Power_on` becomes active, `First` also becomes active. `Shift` is defined as an **Input from Simulink** data object. When `Shift` equals 1, the system transitions from `First` to `Second`, `Second` to `Third`, `Third` to `First`, and then the pattern repeats.

In the Simulink model, there is an event input and a data input. A Sine wave block is used to generate a repeating input event that corresponds with the Stateflow event `Switch`. The Step block is used to generate a repeating pattern of 1 and 0 that corresponds with the Stateflow data object `Shift`. Ideally, the `Switch` event occurs in a frequency that allows at least one cycle through `First`, `Second`, and `Third`.

Typical Scenario to Debug Runtime Errors

These steps describe a typical debugging scenario to resolve runtime errors in the example model:

1. Create the Simulink model and Stateflow diagram (including defining the event and data objects).
2. Ensure the `sfun` target includes debugging options.
3. Invoke the Debugger and choose debugging options.
4. Start the simulation.
5. Debug the simulation execution.
6. Resolve runtime error, and repeat from step 3.

Create the Model and Stateflow Diagram

Using the sample (see “Example Stateflow Diagram” on page 10-12) as a guide, create the Simulink model and Stateflow diagram. Using the graphics editor **Add** menu, add the `Switch` **Input from Simulink** event and the `Shift` **Input from Simulink** data object.

Define the sfun Target

Choose **Open Simulation Target** from the **Tools** menu of the graphics editor. Ensure that the checkbox to **Enable Debugging/Animation** is enabled (checked). Click on the **Close** button to apply the changes and close the dialog box.

Invoke the Debugger and Choose Debugging Options

Choose **Debug** from the **Tools** menu of the graphics editor. Click on the **Chart entry** option under the **Break Controls** border. When the simulation begins, it will break on the entry into the chart. Click on the **Enabled** radio button under the **Animation** border to turn animation on.

Start the Simulation

Click on the **Go** button to start the simulation. Informational messages are displayed in the MATLAB command window. The graphics editor toolbar and menus change appearance to indicate a read-only interface. The Stateflow diagram is parsed, the code is generated, and the target is built. Because you have specified a breakpoint on chart entry, the execution stops at that point and the Debugger display status indicates:

```
Stopped: Just after entering during function of Chart debug__power

Executing: sf_debug_ex_debug_power

Current Event: Input event Switch
```

Debug the Simulation Execution

At this point, you can single step through the simulation and see whether the behavior is what you expect. Click on the **Step** button and watch the Stateflow diagram animation and the Debugger status area to see the sequence of execution.

Single stepping shows that the desired behavior is not occurring. The occurrence of the event input `Switch` is occurring faster than the data input `Shift`. Thus the transitions from `Power_on.First` to `Power_on.Second`, etc are not occurring. The output display of code coverage also confirms this observation.

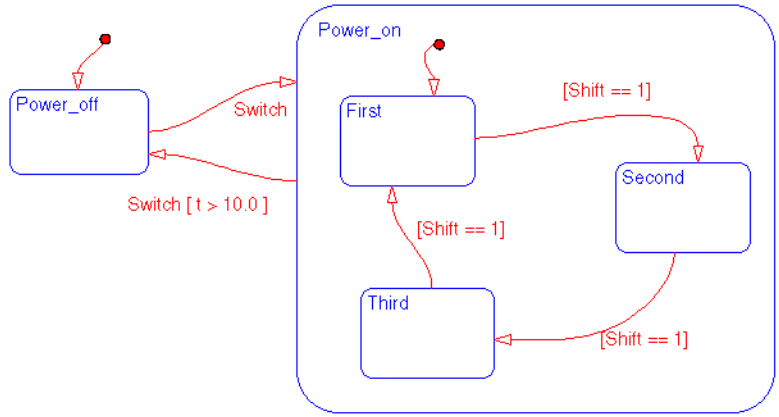
Resolve Runtime Error and Repeat

Choose **Stop** from the **Simulation** menu of the graphics editor. The Stateflow diagram is now writeable. The generation of event `Switch` is driving the simulation and the simulation time is passing too quickly for the input data object `Shift` to have an effect. The model may need to be completely rethought.

In the meantime, there is a test that verifies the conclusion. Modify the transition from `Power_on` to `Power_off` to include a condition. The transition is not to be taken until simulation time is greater than 10.0. Make this modification and click on the **Go** button to start the simulation again. Repeat the debugging single stepping and observe the behavior.

Solution Stateflow Diagram

This is the corrected Stateflow diagram with the condition added to the transition from `Power_on` to `Power_off`.



Debugging State Inconsistencies

What is a State Inconsistency?

Stateflow notations specify that states are consistent if:

- An active state (consisting of at least one substate) with XOR decomposition has exactly one active substate
- All substates of an active state with AND decomposition are active
- All substates of an inactive state with either XOR or AND decomposition are inactive

A state inconsistency error has occured, if at any step in the Stateflow diagram simulation, these notation rules are violated.

Causes of State Inconsistency

State inconsistency errors are most commonly caused by the omission of a default transition to a substate in superstates with XOR decomposition.

Design errors in complex Stateflow diagrams can also result in state inconsistency errors. These errors may only be detectable using the Debugger at runtime.

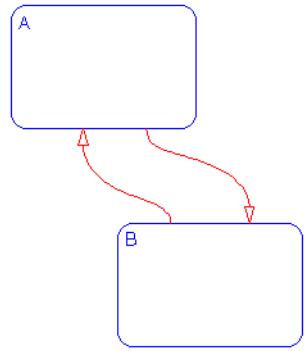
Detecting State Inconsistency

To detect the state inconsistency during a simulation:

- Enable **Debugging/Animation** and the **State Inconsistency** check in the **Target Configuration sfun property** dialog box
- Build the target with the debugging options
- Invoke the Debugger and enable **State Inconsistency** checking
- Start the simulation

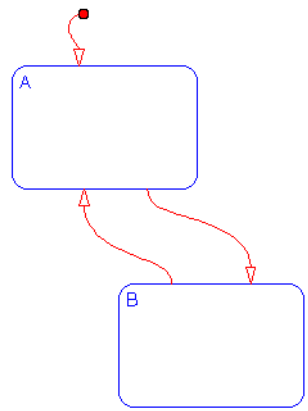
Example: State Inconsistency

This Stateflow diagram has a state inconsistency.



In the absence of a default transition indicating which substate is to become active, the simulation encounters a runtime state inconsistency error.

Adding a default transition to one of the substates resolves the state inconsistency.



Debugging Conflicting Transitions

What is a Conflicting Transition?

A transition conflict exists if, at any step in the simulation, there are two equally valid transition paths from a source to a destination. In the case of a conflict, equivalent transitions (based on their labels) are evaluated based on the geometry of the outgoing transitions. See “Execution Order” on page 7-62 for more information.

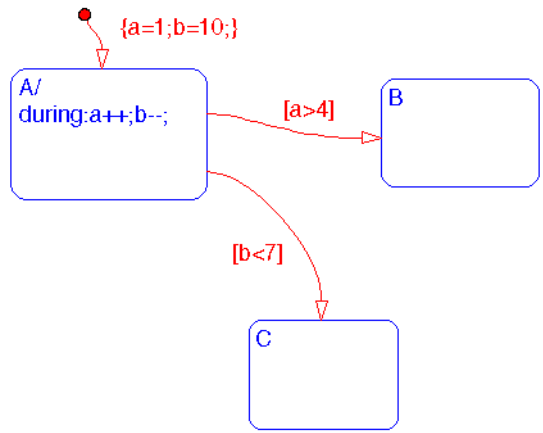
Detecting Conflicting Transitions

To detect conflicting transitions during a simulation:

- Enable **Debugging/Animation** and the **Transition Conflict** check in the **Target Configuration sfun property** dialog box
- Build the target with the debugging options
- Invoke the Debugger and enable **Transition Conflict** checking
- Start the simulation

Example: Conflicting Transition

This Stateflow diagram has a conflicting transition.



The default transition to state A assigns data a equal to 1 and data b equal to ten. State A's during action increments a and decrements b. The transition from state A to state B is valid if the condition [a > 4] is true. The transition from state A to state C is valid if the condition [b < 7] is true. As the simulation proceeds, there is a point where state A is active and both conditions are true. This is a transition conflict.

Multiple outgoing transitions from states that are of equivalent label priority are evaluated in a clockwise progression starting from the 12 o'clock position on the state. In this example, the transition from state A to state B is taken.

Although the geometry is used to continue after the transition conflict, it is not recommended to design your Stateflow diagram based on an expected execution order.

Debugging Data Range Violations

What is a Data Range Violation?

Each Data property dialog box has fields for an **Initial**, **Minimum**, and **Maximum** value. If the data object equals a value outside of this range, enabling data range checking will detect the error.

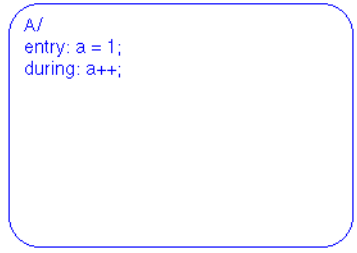
Detecting Data Range Violations

To detect data range violations during a simulation:

- Enable **Debugging/Animation** and **Data Range Checking** in the **Target Configuration sfun property** dialog box
- Build the target with the debugging options
- Invoke the Debugger and enable **Data Range** checking
- Start the simulation

Example: Data Range Violation

This Stateflow diagram has a data range violation.



The data a is defined to have an **Initial** and **Minimal** value of 0 and a **Maximum** value of 2. Each time an event awakens this Stateflow diagram and state A is active, a is incremented. The value of a quickly becomes a data range violation.

Debugging Cyclic Behavior

What is Cyclic Behavior?

When a step or sequence of steps is indefinitely repeated (recursive), this is called cyclic behavior. The Debugger cycle detection algorithms detect a class of infinite recursions caused by event broadcasts.

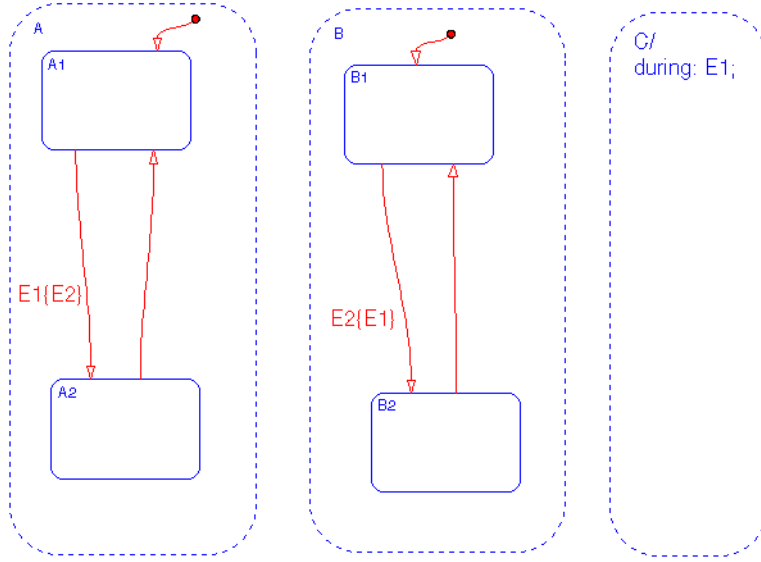
Detecting Cyclic Behavior

To detect cyclic behavior during a simulation:

- Enable **Debugging/Animation** in the **Target Configuration sfun property** dialog box
- Build the target with debugging options
- Invoke the Debugger and enable **Detect Cycles**
- Start the simulation

Example: Cyclic Behavior

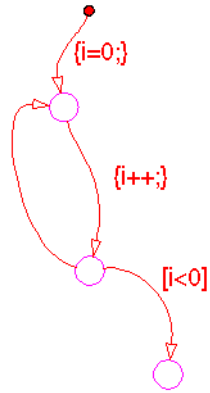
This Stateflow diagram shows a typical example of a cycle created by infinite recursions caused by an event broadcast.



When state `C` `during` action executes event `E1` is broadcast. The transition from state `A.A1` to state `A.A2` becomes valid when event `E1` is broadcast. Event `E2` is broadcast as a condition action of that transition. The transition from state `B.B1` to state `B.B2` becomes valid when event `E2` is broadcast. Event `E1` is broadcast as a condition action of the transition from state `B.B1` to state `B.B2`. Because these event broadcasts of `E1` and `E2` are in condition actions, a recursive event broadcast situation occurs. Neither transition can complete.

Example: Flow Cyclic Behavior Not Detected

This Stateflow diagram shows an example of cyclic behavior in a flow diagram that is not detected by the Debugger.

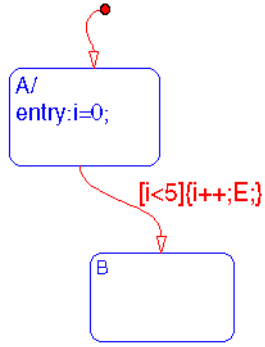


The data object `i` is set to zero in the condition action of the default transition. `i` is incremented in the next transition segment condition action. The transition to the third connective junction is valid only when the condition `[i < 0]` is true. This condition will never be true in this flow diagram and there is a cycle.

This cycle is not detected by the Debugger because it does not involve event broadcast recursion. Detecting cycles that are involved with data values is not currently supported.

Example: Noncyclic Behavior Flagged as a Cycle

This Stateflow diagram shows an example of noncyclic behavior that the Debugger flags as being cyclic.



State `A` becomes active and `i` is initialized to zero. When the transition is tested, the condition `[i < 5]` is true. The condition actions, increment `i` and broadcast event `E`, are executed. The broadcast of `E` when state `A` is active causes a repetitive testing (and incrementing of `i`) until the condition is no longer true. The Debugger flags this as a cycle when in reality the apparent cycle is broken when `i` becomes greater than 5.

11

Reference

This chapter contains detailed descriptions of Stateflow functions.

These functions operate on the machine:

Functions	
`sfnew`	Creates and displays a new Simulink model containing an empty Stateflow block. See `sfnew`.
`sfexit`	Closes all Stateflow diagrams, Simulink models containing Stateflow diagrams, and exits the Stateflow environment. See `sfexit`.
`sfsave`	Saves the current machine and Simulink model. See `sfsave`.
`stateflow`	Opens the Stateflow model window. See `stateflow`.

This function operates on a Stateflow diagram:

Functions	
`sfprint`	Prints the visible portion of a Stateflow diagram. See `sfprint`.

This function is independent of models and Stateflow diagrams.

Functions	
`sfhelp`	Displays Stateflow online help in an HTML browser. See `sfhelp`.

The Stateflow model window contains an empty Stateflow block, an Examples block , and a manual switch. A Stateflow block is equivalent to a Stateflow diagram.

Purpose

Create a new Simulink model containing an empty Stateflow block.

Syntax

```
sfnew
sfnew modelname
```

Description

`sfnew` creates and displays a new untitled Simulink model containing an empty Stateflow block.

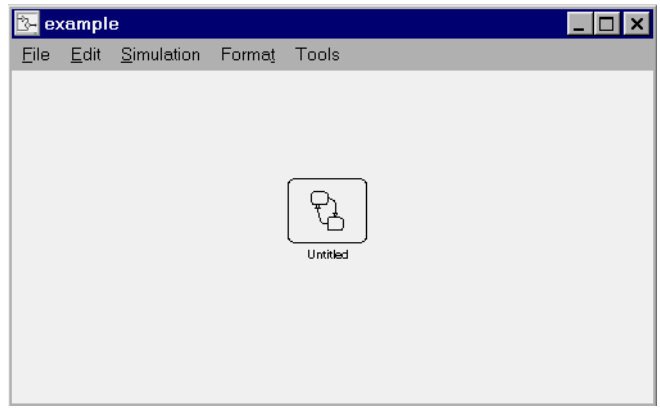
`sfnew` *modelname* creates and displays a new Simulink model with the title specified.

Example

Create and display a new titled Simulink model, which contains an empty Stateflow block:

```
>>sfnew example
```

A new Simulink model, titled example, which contains an empty Stateflow block, is displayed.



sfexit

Purpose

Close all Stateflow diagrams, Simulink models containing Stateflow diagrams, and exit the Stateflow environment.

Syntax

```
sfexit
```

Description

`sfexit` closes all Stateflow diagrams, all Simulink models containing Stateflow diagrams, and exits the Stateflow environment.

`sfexit` unlocks the Stateflow MEX-file and clears it from the workspace.

Purpose

Save the current machine and Simulink model.

Syntax

```
sfsave
sfsave ('machinename')
sfsave ('machine', 'saveasname')
sfsave ('defaults')
```

Description

`sfsave` saves the current machine and Simulink model.

`sfsave ('machinename')`saves the specified machine and its Simulink model.

`sfsave ('machine', 'saveasname')`saves the specified machine and its Simulink model with the specified name.

`sfsave ('defaults')`saves the current environement default settings in the defaults file.

stateflow

Purpose

Open the Stateflow model window.

Syntax

```
stateflow
```

Description

`stateflow` opens the Stateflow model window. The model contains an untitled Stateflow block, an Examples block, and a manual switch. The Stateflow block is a masked Simulink model and is equivalent to an empty, untitled Stateflow diagram. Use the Stateflow block to include a Stateflow diagram in a Simulink model.

Every Stateflow block has a corresponding S-function. This S-function is the agent Simulink interacts with for simulation and analysis.

The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow blocks into Simulink models, you can add event-driven behavior to Simulink simulations. You create models that represent both data and control flow by combining Stateflow blocks with the standard Simulink and toolbox blocksets. These combined models are simulated using Simulink.

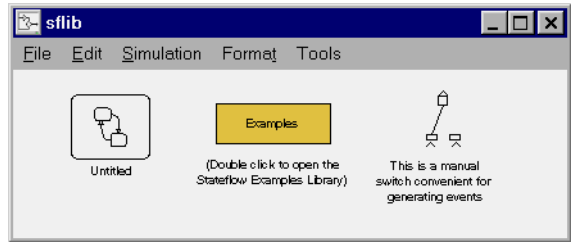
Example

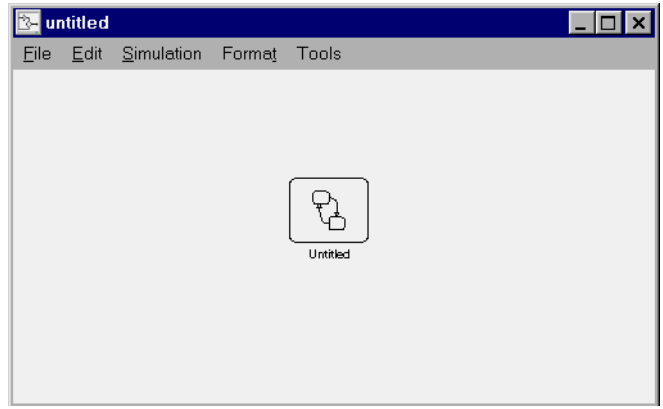
This example shows how to open the Stateflow model window and use a Stateflow block to create a Simulink model.

1 Invoke Stateflow:

```
>>stateflow
```

The Stateflow model window and an untitled Simulink model containing a Stateflow block are displayed:


sflib
File Edit Simulation Format Tools
Untitled
Examples
(Double click to open the Stateflow Examples Library)
This is a manual switch convenient for generating events



2 Double-click on the untitled Stateflow block in the untitled Simulink model to invoke a Stateflow editor window.

3 You are ready to create the underlying Stateflow diagram.

Purpose

Print the visible portion of the Stateflow diagram.

Syntax

```
sfprint
sfprint ('diagram_name','ps')
sfprint ('diagram_name','psc')
sfprint ('diagram_name','tif')
sfprint ('diagram_name','clipboard')
```

Description

`sfprint` prints the visible portion of the current Stateflow diagram. A read-only preview window appears while the print operation is in progress. An informational box appears indicating the printing operation is starting.

See "Printing Large Stateflow Diagrams" on page 2-23, for information on printing Stateflow diagrams that are larger than the editor display area.

`sfprint ('diagram_name','ps')` prints the visible portion of the specified Stateflow diagram to a postscript file.

`sfprint ('diagram_name','psc')` prints the visible portion of the specified Stateflow diagram to a color postscript file.

`sfprint ('diagram_name','tif')` prints the visible portion of the specified Stateflow diagram to a `.tif` file.

`sfprint ('diagram_name','clipboard')` prints the visible portion of the specified Stateflow diagram to a clipboard bitmap (PC version only).

sfhelp

Purpose

Display Stateflow online help in an HTML browser.

Syntax

```
sfhelp
```

Description

`sfhelp` invokes your HTML browser (Netscape Navigator, Internet Explorer, etc.) displaying the Stateflow online help. An online version of the *Stateflow User's Guide* is also accessible from the online help.

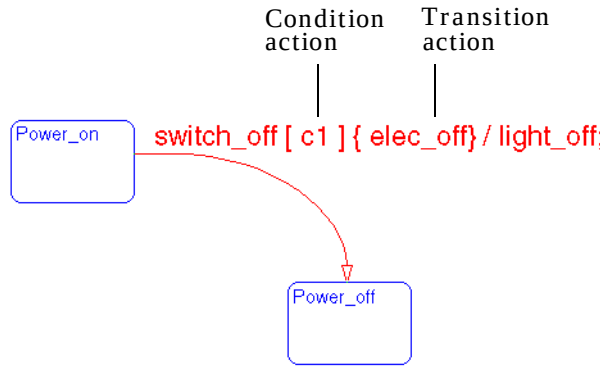
Your HTML browser must support HTML frames.

B

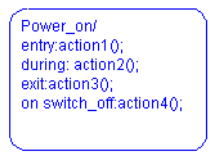
Glossary

Actions

Actions take place as part of Stateflow diagram execution. The action can be executed as part of a transition from one state to another, or depending on the activity status of a state. Transitions can have condition actions and transition actions. For example:



States can have `entry`, `during`, `exit`, and, `on` *`event_name`* actions. For example:

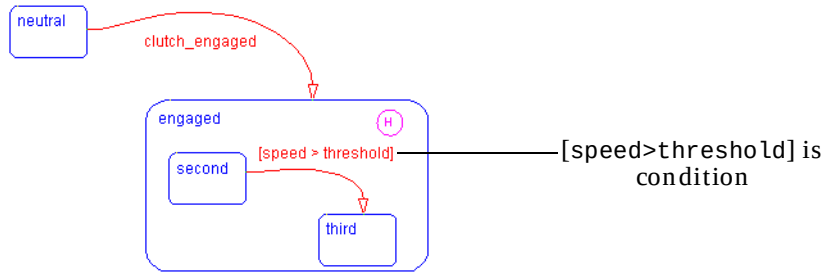


If you enter the name and backslash followed directly by an action or actions (without the `entry` keyword), the action(s) are interpreted as `entry` action(s). This shorthand is useful if you are only specifying `entry` actions.

The *action language* defines the categories of actions you can specify and their associated notations. An action can be a function call, an event to be broadcast, a variable to be assigned a value, etc. For more information, see the section titled “Action Language” on page 6-42.

Condition

A *condition* is a Boolean expression to specify that a transition occurs given that the specified expression is true. For example,

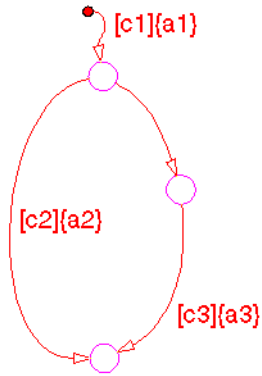


The action language defines the notation to define conditions associated with transitions. See the section titled “Action Language” on page 6-42 for more information.

Connective Junction

Connective junctions are decision points in the system. A connective junction is a graphical object that simplifies Stateflow diagram representations and facilitates generation of efficient code. Connective junctions provide alternative ways to represent desired system behavior.

This example shows how connective junctions (displayed as small circles) are used to represent the flow of an `if` code structure.




```
if [c1]{
      a1
      if [c2]{
         a2
      }else if [c3]{
        a3
      }
}
```

Name	Button Icon	Description
Connective junction		One use of a Connective junction is to handle situations where transitions out of one state into two or more states are taken based on the same event but guarded by different conditions.

See the section titled “Connective Junctions” on page 6-32 for more information.

Data

Data objects store numerical values for reference in the Stateflow diagram.

See “Data” on page 3-22 for more information on representing data objects.

Data Dictionary

The *data dictionary* is a database where Stateflow diagram information is stored. When you create Stateflow diagram objects, the information about those objects is stored in the data dictionary once you save the Stateflow diagram.

Debugger

See “Stateflow Debugger”.

Decomposition

A state has a *decomposition* when it consists of one or more substates. A Stateflow diagram that contains at least one state also has decomposition. Representing hierarchy necessitates some rules around how states can be grouped in the hierarchy. A superstate has either parallel (AND) or exclusive (OR) decomposition. All substates at a particular level in the hierarchy must be of the same decomposition.

Parallel (AND) State Decomposition. Parallel (AND) state decomposition is indicated when states have dashed borders. This representation is appropriate if all states at that same level in the hierarchy are active at the same time. The activity within parallel states is essentially independent.

Exclusive (OR) State Decomposition. Exclusive (OR) state decomposition is represented by states with solid borders. Exclusive (OR) decomposition is used to describe system modes that are mutually exclusive. Only one state, at the same level in the hierarchy, can be active at a time.

Default Transition

Default transitions are primarily used to specify which exclusive (OR) state is to be entered when there is ambiguity among two or more neighboring exclusive (OR) states. For example, default transitions specify which substate of a superstate with exclusive (OR) decomposition the system enters by default in the absence of any other information. Default transitions are also used to specify that a junction should be entered by default. A default transition is represented by selecting the default transition object from the toolbar and then dropping it to attach to a destination object. The default transition object is a transition with a destination but no source object.

Name	Button Icon	Description
Default transition		Use a Default transition to indicate, when entering this level in the hierarchy, which state becomes active by default.

See the section titled “What is a Default Transition?” on page 6-24 for more information.

Events

Events drive the Stateflow diagram execution. All events that affect the Stateflow diagram must be defined. The occurrence of an event causes the status of the states in the Stateflow diagram to be evaluated. The broadcast of an event can trigger a transition to occur and/or can trigger an action to be executed. Events are broadcast in a top-down manner starting from the event's parent in the hierarchy.

Events are added, removed and edited through the Stateflow Explorer. See the section titled “Events” on page 3-5 for more information.

Explorer

See “Stateflow Explorer”.

Finder

See “Stateflow Finder”.

Finite State Machine

A *finite state machine* (*FSM*) is a representation of an event-driven system. FSMs are also used to describe reactive systems. In an event-driven or reactive system, the system transitions from one mode or state, to another prescribed mode or state, provided that the condition defining the change is true.

Hierarchy

Hierarchy enables you to organize complex systems by placing states within other higher-level states. A hierarchical design usually reduces the number of transitions and produces neat, more manageable diagrams. See the section titled “Hierarchy” on page 1-15 for more information.

History Junction

A *History junction* provides the means to specify the destination substate of a transition based on historical information. If a superstate has a History junction, the transition to the destination substate is defined to be the substate

that was most recently visited. The History junction applies to the level of the hierarchy in which it appears.

Name	Button Icon	Description
History junction		Use a History junction to indicate, when entering this level in the hierarchy, that the last state that was active becomes the next state to be active.

See these sections for more information:

- "What Is a History Junction?" on page 6-40
- "Example: Default Transition and a History Junction" on page 7-21
- "Example: Labeled Default Transitions" on page 7-23
- "Example: Inner Transition to a History Junction" on page 7-31

Inner Transitions

An *inner transition* is a transition that does not exit the source state. Inner transitions are most powerful when defined for superstates with XOR decomposition. Use of inner transitions can greatly simplify a Stateflow diagram.

See the sections titled "What Is an Inner Transition?" on page 6-27 and "Example: Inner Transition to a History Junction" on page 7-31 for more information.

Machine

The *machine* is the collection of all Stateflow blocks in the Simulink model.

Notation

A *notation* defines a set of objects and the rules that govern the relationships between those objects. Stateflow notation provides a common language to communicate the design information conveyed by a Stateflow diagram.

Stateflow notation consists of:

- A set of graphical objects
- A set of nongraphical text-based objects
- Defined relationships between those objects

Parallelism

A system with *parallelism* can have two or more states that can be active at the same time. The activity of parallel states is essentially independent. Parallelism is represented with a parallel (AND) state decomposition.

See the section titled “What Is a State Decomposition?” on page 6-10 for more information.

Real-Time Workshop

The Real-Time Workshop is an automatic C language code generator for Simulink. It produces C code directly from Simulink block diagram models and automatically builds programs that can be run in real-time in a variety of environments.

See the *Real-Time Workshop User's Guide* for more information.

S-function

When using Simulink together with Stateflow for simulation, Stateflow generates an *S-function* (MEX-file) for each Stateflow machine to support model simulation. This generated code is a simulation target and is called the `sfun` target within Stateflow.

For more information, see the *Using Simulink* document.

Semantics

Semantics describe how the notation is interpreted and implemented behind the scenes. A completed Stateflow diagram communicates how the system will behave. A Stateflow diagram contains actions associated with transitions and states. The semantics describe in what sequence these actions take place during Stateflow diagram execution.

Simulink

Simulink is a software package for modeling, simulating, and analyzing dynamic systems. It supports linear and nonlinear systems, modeled in continuous time, sampled time, or a hybrid of the two. Systems can also be multirate, i.e., have different parts that are sampled or updated at different rates.

It allows you to represent systems as block diagrams that you build using your mouse to connect blocks and your keyboard to edit block parameters. Stateflow is part of this environment. The Stateflow block is a masked Simulink model. Stateflow builds an S-function that corresponds to each Stateflow machine. This S-function is the agent Simulink interacts with for simulation and analysis.

The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow diagrams into Simulink models, you can add event-driven behavior to Simulink simulations. You create models that represent both data and control flow by combining Stateflow blocks with the standard Simulink blocksets. These combined models are simulated using Simulink.

The *Using Simulink* document describes how to work with Simulink. It explains how to manipulate Simulink blocks, access block parameters, and connect blocks to build models. It also provides reference descriptions of each block in the standard Simulink libraries.

State

A *state* describes a mode of a reactive system. A reactive system has many possible states. States in a Stateflow diagram represent these modes. The activity or inactivity of the states dynamically changes based on events and conditions.

Every state has hierarchy. In a Stateflow diagram consisting of a single state, that state's parent is the Stateflow diagram itself. A state also has history that applies to its level of hierarchy in the Stateflow diagram. States can have

actions that are executed in a sequence based upon action type. The action types are: `entry`, `during`, `exit`, or on *event_name* actions.

Name	Button Icon	Description
State		Use a state to depict a mode of the system.

Stateflow Block

The *Stateflow block* is a masked Simulink model and is equivalent to an empty, untitled Stateflow diagram. Use the Stateflow block to include a Stateflow diagram in a Simulink model.

The control behavior that Stateflow models complements the algorithmic behavior modeled in Simulink block diagrams. By incorporating Stateflow blocks into Simulink models, you can add complex event-driven behavior to Simulink simulations. You create models that represent both data and control flow by combining Stateflow blocks with the standard Simulink and toolbox block libraries. These combined models are simulated using Simulink.

Stateflow Debugger

Use the *Stateflow Debugger* to debug and animate your Stateflow diagrams. Each state in the Stateflow diagram simulation is evaluated for overall code coverage. This coverage analysis is done automatically when the target is compiled and built with the debug options. The Debugger can also be used to perform dynamic checking. The Debugger operates on the Stateflow machine.

Stateflow Diagram

Using Stateflow, you create Stateflow diagrams. A *Stateflow diagram* is also a graphical representation of a finite state machine where *states* and *transitions* form the basic building blocks of the system. See the section titled “Anatomy of a Model and Machine” on page 1-8 for more information on Stateflow diagrams.

Stateflow Explorer

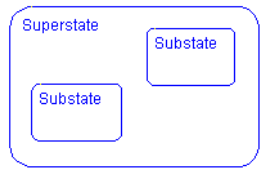
Use the *Explorer* to add, remove, and modify data, event, and target objects. See, “Stateflow Explorer User Interface” on page 5-5 for more information.

Stateflow Finder

Use the *Finder* to display a list of objects based on search criteria you specify. You can directly access the properties dialog box of any object in the search output display by clicking on that object. See “Stateflow Finder User Interface” on page 5-20 for more information.

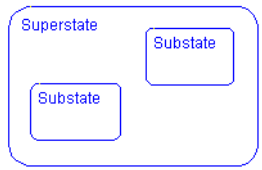
Substate

A state is a *substate* if it is contained by a superstate.



Superstate

A state is a *superstate* if it contains other states, called substates.



Transition

A *transition* describes the circumstances under which the system moves from one state to another. Either end of a transition can be attached to a source and a destination object. The *source* is where the transition begins and the *destination* is where the transition ends. It is often the occurrence of some event that causes a transition to take place.

Virtual Scrollbar

A *virtual scrollbar* enables you to set a value by scrolling through a list of choices. When you move the mouse over a menu item with a virtual scrollbar, the cursor changes to a line with a double arrowhead. Virtual scrollbars are either vertical or horizontal. The direction is indicated by the positioning of the

arrowheads. Drag the mouse either horizontally or vertically to change the value.

See the section titled “Virtual Scrollbar” on page 2-7 for more information.

Index

A

action language
  array arguments 6-56
  assignment operations 6-48
  binary operations 6-45
  bit operations 6-44
  choosing between `ml()` and `ml.` 6-55
  comments 6-62
  components 6-44
  continuation symbols 6-62
  data and event arguments 6-56
  definition 1-19
  directed event broadcasting 6-58
  event broadcasting 6-56
  glossary definition A-2
  literals 6-62
  objects with actions 6-42
  state action notation 6-43
  time symbol 6-61
  transition action notation 6-43
  unary operations 6-47, 6-48
  use of the semicolon 6-62
  user-written functions 6-49
actions
  definition 1-18
  glossary definition A-2
animation
  debugger control 10-10
  enable 9-24
array arguments 6-56
assignment operations 6-48

B

binary operations 6-45
bit operations 6-44
breakpoints
  global 10-3
    chart entry 10-8
    event broadcast 10-8
    state entry 10-8
  local 10-3
  overview 10-3

C

C compiler requirements xxii
canceling operations 2-15
carousel
  run the demo 8-27
  run the provided solution 8-24
  Simulink model solution 8-24
  solution
    procedure to run the simulation 8-25
    Stateflow diagram 8-25
  tutorial tasks 8-8
chapter quick reference xxvi
chart (Stateflow diagram)
  debugger breakpoint property 2-20
  description property 2-20
  document link 2-20
  editing chart properties 2-19
  editor property 2-20
  hyperlinks example 2-21
  name property 2-20
  operations 2-18
  overview 2-18
  parent property 2-20
  printing, large 2-23
  sample time property 2-20
  Simulink subsystem property 2-20
  update method property 2-20
  update methods 2-22
  update methods for defining interface 4-4

code generation
adding and specifying a stand-alone target 9-16
coder file 9-5
error messages 9-32
notations to optimize code size 9-8
notations to optimize memory use 9-9
notations to optimize speed 9-7
stand-alone target 9-16
typical sequence 9-5
typical tasks 9-4
using notation efficiently 9-7
code generation related error messages 9-30
coder command customization 9-23
coder file 9-5
combination state
definition 6-12
notation 6-12
compilation error messages 9-32
condition
definition 1-16
glossary definition A-3
notation overview 6-61
condition action
actions specified as condition actions semantics example 7-13
actions specified as condition and transition actions semantics example 7-14
cyclic behavior to avoid semantics example 7-17
using condition actions in `for` loop construct semantics example 7-15
conflicting transitions 10-18
connective junction
common destination notation example 6-38
common events notation example 6-38
definition 1-21
flow diagram semantics example 7-37
`for` loop construct semantics example 7-35
`for` loops notation example 6-35
from a common source notation example 6-37
glossary definition A-3
`if-then-else` decision construct semantics example 7-33
notation overview 6-32
self loop notation example 6-34
self loop semantics example 7-34
transitions based on a common event semantics example 7-41
transitions from a common source to multiple destinations semantics example 7-39
transitions from multiple sources semantics example 7-40
with all conditions specified notation example 6-33
with one unconditional transition notation example 6-33
continuation symbols 6-62
custom code customization 9-25
cycle detection 10-21

D

data
  attribute properties
    initial value 3-34
    maximum value 3-34
    minimum value 3-34
    type 3-34
    units 3-33
  constant 3-31
  definition 1-15
  description property 3-24
  exported 4-28
  exported to an external code source 3-31
  getting started 3-23
  getting started example 5-16
  glossary definition A-4
  imported 4-29
  imported from an external code source 3-31
  input from Simulink 3-27
  local 3-27
  name property 3-24
  output to Simulink 3-28
  parent property 3-25
  port property 3-32
  properties 3-23
    properties dialog box 5-12
      apply button 5-15
      close button 5-15
      description field 5-14
      document link field 5-14
      initial value 5-14
      maximum field 5-14
      minimum field 5-14
      name field 5-12
      parent field 5-12
      port field 5-13
      revert button 5-15
      scope field 5-12
      type field 5-14
      units field 5-12
    scope property 3-27
    temporary 3-29
    what is a data object 3-22
    workspace 3-30
data attributes 3-33
data dictionary
  definition 6-4
  glossary definition A-4
data input from Simulink
  add and choose a parent task 4-18
  apply and save task 4-19
  choose scope task 4-18
  specify data attributes task 4-19
  tasks 4-18
data output to Simulink
  add and choose a parent task 4-22
  apply and save task 4-23
  choose scope task 4-22
  specify data attributes task 4-23
  tasks 4-22
data range violation 10-20
data scope and parent combinations 3-32

data, hierarchy, and scope 3-26
Debugger
  action control buttons 10-9
  active states display 10-11
  animation controls 10-10
  break button 10-9
  break on controls 10-8
  breakpoints 10-3
  breakpoints display 10-11
  browse data display 10-11
  call stack display 10-11
  clear output display 10-11
  conflicting transitions
    definition 10-18
    detecting 10-18
    example 10-18
  coverage display 10-11
  cyclic behavior
    definition 10-21
    detecting 10-21
    example 10-22
  data range violation
    definition 10-20
    detecting 10-20
    example 10-20
  debugging conflicting transitions 10-18
  debugging cycle detection 10-21
  debugging data range violations 10-20
debugging runtime errors example 10-12
  choose options task 10-14
  create model task 10-13
  debug simulation task 10-14
  define sfun task 10-14
  example Stateflow diagram 10-12
  resolve errors task 10-15
  solution Stateflow diagram 10-15
  start simulation task 10-14
  typical tasks 10-13
debugging state inconsistencies 10-16
display controls 10-11
dynamic check
  cyclic behavior 10-10
  data range violation 10-10
  state inconsistency 10-10
  transition conflict 10-10
Dynamic Checker options 10-10
flow cyclic behavior
  example 10-23
glossary definition A-10
go button 10-9
including in the target build 10-3
main window 10-6
MATLAB command field 10-11
overview 10-2
runtime debugging 10-4
state inconsistency
  causes 10-16
  definition 10-16
  detecting 10-16
  example 10-17
status display area 10-8
step button 10-9
stop simulation 10-9
typical tasks 10-3
user interface 10-6

debugging
enable 9-24
overview 10-2
runtime 10-4
decomposition
glossary definition A-5
specifying 2-24
default transition
and a history junction semantics example 7-21
creating 2-32
definition 1-20
glossary definition A-5
in an exclusive (OR) decomposition semantics example 7-18
labeled default transition semantics example 7-23
notation example 6-25
notation labeling 6-24
notation overview 6-24
to a junction notation example 6-26
to a junction semantics example 7-20
with a label notation example 6-26
defaults
chart 2-51
colors 2-51
data 2-54
event 2-54
how to change 2-50
junction 2-53
machine 2-50
state 2-53
target 2-52
transition 2-53
defaults file overview 2-50
directed event broadcasting
send semantics example 7-58
using qualified event names notation example 6-60
using qualified event names semantics example 7-60
using send notation example 6-59
document link customization 9-25
during keyword 2-26

E

edge trigger type
either edge 3-18
falling edge 3-18
rising edge 3-18
edge triggered
defining output events 4-13
example output event semantics 4-13
either edge triggered 3-18
entry keyword 2-26
error messages
code generation 9-32
code generation related 9-30
compilation 9-32
parsing 9-30
target building 9-32

event
broadcasting 6-56
definition 1-14
description property 3-7
directed event broadcast using qualified names 6-60
directed event broadcasting 6-58
exported 4-25
exported to an external code source 3-13
imported 4-27
imported from an external code source 3-14
index property 3-16
input from Simulink 3-10
local 3-10
name property 3-6
output to Simulink 3-12
parent and scope combinations 3-15
parent property 3-7
port property 3-16
properties 3-5
properties dialog box 5-9
apply button 5-11
close button 5-11
debugger breakpoint fields 5-11
description field 5-11
document link field 5-11
index field 5-10
name field 5-9
parent field 5-9
port field 5-10
revert button 5-11
scope field 5-9
trigger field 5-10
representing hierarchy 6-9
trigger property 3-17
what is 3-5
event actions and superstates semantics example 7-43
event broadcast
state action notation example 6-57
transition action notation example 6-57
event input from Simulink
add and choose parent task 4-16
apply the changes task 4-17
choose scope task 4-16
select the trigger task 4-17
tasks 4-16
trigger type, determining 3-18
event output to Simulink
add and choose a parent task 4-20
apply and save task 4-21
choose scope task 4-20
task overview 4-10
tasks 4-20
event triggers
defining edge triggered 4-13
defining function call 4-10
defining output to Simulink 4-8
edge triggered semantics example 4-13
function call example 4-11
function call semantics 4-12
events
getting started 3-6
getting started example 5-16
glossary definition A-6
how are they broadcast 3-6
input from Simulink
function call and edge triggers 3-19
scope property 3-10

events and data
  getting started
    add local events task 5-18
    adding local data task 5-19
    what are the events task 5-17
    what is the data task 5-18
  getting started example 5-16
  getting started solution 5-16
  getting started task overview 5-16
events, hierarchy, and scope 3-9
exclusive (OR) state notation 6-11
exit keyword 2-26
Explorer
  add menu 5-7
  contents of list 5-6
  deleting objects from 5-8
  edit menu 5-6
  file menu 5-6
  help menu 5-7
  main window 5-5
  moving objects to change index and port order 5-8
  moving objects to change parents 5-8
  object hierarchy list 5-5
  overview 5-2
  typical tasks 5-3
  user interface 5-5
exploring and searching overview 5-2
exported
  data 4-28
  event 4-25
external code sources
  defining interface for 4-25
  definition 4-25

F

falling edge triggered 3-18
Finder
  dialog box 5-20
    clear button 5-22
    close button 5-22
    display area 5-23
    help button 5-22
    matches field 5-22
    object type 5-21
    refine button 5-22
    representing hierarchy 5-24
    search button 5-22
    search history list 5-22
    search method 5-21
    string criteria field 5-20
  glossary definition A-11
  overview 5-2
  typical tasks 5-4
  user interface 5-20
finite state machine
  glossary definition A-6
  references 1-7
  representations 1-6
  what is 1-6
flow diagram
  for loop notation example 6-35
  notation example 6-36
  notation optimization 9-9
  notation overview 6-32
  overview example 6-36
for loop
  notation example 6-35
  semantics example 7-35

function call
combining with edge triggers 3-19
defining output event 4-10
example output event semantics 4-12
output event example 4-11
function call and edge triggers, combining 3-19
function call triggered 3-18
functions
sfexit 11-4
sfhelp 11-10
sfnew 11-3
sfprint 11-9
sfsave 11-5
stateflow 11-6
user-written 6-49

G

graphical objects
copying 2-16
cutting and pasting 2-16
selecting and deselecting 2-15
graphics editor
add menu 2-48
drawing area 2-3
edit menu 2-45
file menu 2-43
help menu 2-49
main window components overview 2-2
menu bar 2-5
menus overview 2-42
message area 2-3
modified flag 2-5
object button modes 2-15
overview 2-2
print as submenu 2-44
productivity boosts 2-7
scroll bars 2-4
shortcut menus overview 2-10
simulation menu 2-45
Stateflow diagram name 2-4
style menu 2-46
toolbar 2-6
tools menu 2-47
using 2-15
virtual scrollbar 2-7
zooming options 2-4

H

hierarchy 1-15
definition 1-15
events 6-9
events and data 3-3
glossary definition A-6
state 6-7
transition 6-8
history junction
and a default transition semantics example 7-21
and an inner transition semantics example 7-31
definition 1-17
glossary definition A-6
notation overview 6-40
use of history junctions notation example 6-40

I

if-then-else
another notation example 6-33
notation example 6-33
semantics example 7-33
implicit local events
example 3-21
overview 3-20
types 3-20
imported
data 4-29
event 4-27
inner transition
before using an inner transition(1) notation example 6-27
before using an inner transition(2) notation example 6-29
glossary definition A-7
notation overview 6-27
processing a second event with an inner transition to a connective junction semantics example 7-29
processing a second event within an exclusive (OR) state semantics example 7-26
processing a third event within an exclusive (OR) state semantics example 7-27
processing one event with an inner transition to a connective junction semantics example 7-28
processing one event within an exclusive (OR) state semantics example 7-25
to a connective junction(1) notation example 6-28
to a connective junction(2) notation example 6-29
to a history junction notation example 6-30
to a history junction semantics example 7-31
installation xxviii

J

junction
changing incoming arrowhead size 2-39
changing size 2-39
creating and deleting 2-38
description property 2-41
document link property 2-41
editing properties 2-40
moving 2-40
operations 2-38
parent property 2-41
properties 2-41

K

keywords
change(data_name) 6-5
during 6-5
entry 6-5
entry(state_name) 6-5
exit 6-5
exit(state_name) 6-6
in(state_name) 6-6
matlab() 6-6
matlab. 6-6
on event_name 6-6
send(event_name,state_name) 6-6
summary list 6-5

L

literals 6-62

M

machine
  glossary definition A-7
  what is 3-8
make command customization 9-23
MATLAB
  overview xxiii
  requirements for xxii
ml() functions 6-50
ml. name space operator 6-53

N

notation
  action language
    array arguments 6-56
    assignment operations 6-48
    binary operations 6-45
    bit operations 6-44
    choosing between ml() and ml. 6-55
    comments 6-62
    components 6-44
    condition overview 6-61
    continuation symbols 6-62
    data and event arguments 6-56
    directed event broadcasting 6-58
    event broadcasting 6-56
    literals 6-62
    objects with actions 6-42
    state action 6-43
    time symbol 6-61
    transition action 6-43
    unary actions 6-48
    unary operations 6-47
    use of semicolon 6-62
    user-written functions 6-49
  connective junction overview 6-32
default transition definition 6-24
default transition example 6-25
default transition labeling 6-24
definition 1-7, 6-2
designing for efficient code 9-7
flow diagram overview 6-32
glossary definition A-7
graphical objects 6-3
history junction overview 6-40
history junctions and inner transitions overview 6-41
how is the notation checked 6-3
inner transition overview 6-27
keywords 6-5
motivation for 6-2
nongraphical objects 6-4
representing hierarchy 6-6
self loop transition overview 6-31
state
  during action 6-15
  entry action 6-14
  exit action 6-15
  labeling example 6-14
  name 6-14
  on action 6-15
transition
  label example 6-18
  labeling 6-18
  valid label 6-19
transition definition 6-17
transition label
  condition 6-18
  condition action 6-19
  event 6-18
  transition action 6-19
transition types 6-20

O

on keyword 2-26

P

parallel (AND) state
- event broadcast condition action semantics example 7-54
- event broadcast state action semantics example 7-45
- event broadcast transition action semantics example 7-50
- notation 6-11

parallelism
- definition 1-19
- glossary definition A-8

parsing
- error messages 9-30
- how to 9-11
- tasks 9-12

prerequisites xxii

Q

quick start
- creating a Simulink model 1-23
- creating a Stateflow diagram 1-26
- debugging the Stateflow diagram 1-33
- defining the Stateflow interface 1-30
- generating code 1-35
- overview 1-22
- running a simulation 1-32
- sample solution 1-22
- Stateflow typical tasks 1-22

R

Real-Time Workshop
- glossary definition A-8
- overview xxv
- requirements for xxii

references 1-7

regular expressions 5-21

rising edge triggered 3-18

`rtw` target
- compiling/building 9-27
- configuration dialog box 9-22

S

searching
- based on multiple conditions example 5-32
- Finder user interface 5-20
- for a wildcard string example 5-26
- for all objects of a specified object type example 5-30
- for an exact string match example 5-28
- for an object to edit properties example 5-34

self loop
- notation example 6-34
- notation overview 6-31

semantics
- definition 1-7
- execution order 7-62
- glossary definition A-8

`send`
- keyword 6-6
- notations example 6-59
- semantics example 7-58

`sfc.cdr` 9-5

`sfexit` 11-4

`sfhelp` 11-10

`sfnew` 11-3

sfprint 11-9
sfsave 11-5
S-function glossary definition A-8
shortcut menus
  general 2-12
  junction 2-14
  options available on all shortcut menus 2-10
  state 2-13
  transition 2-14
simulation menu 2-45
simulation target
  compiling/building tasks 9-27
  configuation dialog box 9-21
Simulink
  glossary definition A-9
  overview xxiv
  requirements for xxii
Simulink model and Stateflow machine
  relationship between 1-8
stand-alone target
  add target task 9-17
  adding and specifying tasks 9-16
  compiling/building 9-28
  configuration dialog box 9-23
state
  active and inactive notation 6-11
  active notation 6-11
  changing incoming transition arrowhead size 2-28
  combination notation 6-12
  creating and deleting 2-25
  debugger breakpoint property 2-30
  decomposition notation 6-10
  definition 1-12
  description property 2-30
  document link property 2-30
  editing properties 2-29
  exclusive (OR) decomposition notation 6-11
  glossary definition A-9
  grouping and ungrouping 2-26
  inactive notation 6-11
  label property 2-30
  labeling 2-26
  labeling notation 6-13
  moving 2-28
  name property 2-30
  notation
    decomposition 6-10
    during action 6-15
    entry action 6-14
    exit action 6-15
    labeling example 6-14
    name 6-14
    on action 6-15
  notation overview 6-10
  operations 2-24
  parallel (AND) decomposition notation 6-11
  parent property 2-30
  properties 2-30
  representing hierarchy 6-7
  resizing 2-27
  showing and hiding 2-27
  specifying decomposition 2-24
  unique name notation example 6-15
state inconsistencies 10-16
state label
  changing font size 2-27
  during keyword 2-26
  entry keyword 2-26
  exit keyword 2-26
  format 2-26
  on keyword 2-26

Stateflow
  applications, types of 1-3
  component overview 1-3
  defining interfaces overview 1-10
  design approaches 1-4
  feature overview 1-2
  representations 1-6
stateflow 11-6
Stateflow block
  considerations in choosing continuous update 4-7
  continuous 4-4
  continuous example 4-8
  defining a continuous 4-7
  defining a sampled 4-5
  defining a triggered 4-5
  defining an inherited 4-6
  inherited 4-4
  inherited example 4-7
  sampled 4-4
  sampled example 4-6
  triggered 4-4
  triggered example 4-5
  update methods 4-4
Stateflow Coder overview 9-2
Stateflow diagram
  glossary definition A-10
  graphical components 1-12
  objects 1-11
  update methods 4-4
Stateflow Dynamic Checker
  data range violations 9-25
  overview 9-3
  state inconsistencies 9-24
  transition conflicts 9-25
Stateflow interfaces
  overview 4-2
  typical tasks to define 4-2
substate glossary definition A-11
superstate glossary definition A-11

T

target
  build options 9-3
  building error messages 9-32
  compile options 9-3
  customizing 9-20
  potential customization 9-20
target configuration sfun dialog box 9-21
target customization
  add custom code 9-25
  coder command 9-23
  document link 9-25
  enable debugging/animation 9-24
  enable Dynamic Checker options 9-24
  make command 9-23
time 6-61

transition
changing arrowhead size 2-34
creating and deleting 2-31
debugger breakpoint property 2-37
debugging conflicting 10-18
default 2-32
definition 1-13
description property 2-37
destination property 2-37
document link property 2-37
editing attach points 2-32
editing properties 2-36
glossary definition A-11
label format 2-33
label property 2-37
labeling 2-33
notation
label example 6-18
labeling 6-18
types 6-20
valid label 6-19
notation definition 6-17
operations 2-31
parent property 2-37
properties 2-37
representing hierarchy 6-8
source property 2-37
to and from exclusive (OR) states(1) semantics example 7-8
to and from exclusive (OR) states(2) semantics example 7-9
to and from exclusive (OR) states(3) semantics example 7-10
to and from junctions notation example 6-21
to and from OR states notation example 6-21
to and from OR superstates notation example 6-22
to and from substates notation example 6-23
valid labels 2-34
transition label
changing font size 2-34
condition 2-33
condition action 2-33
event 2-33
moving 2-35
notation
condition 6-18
condition action 6-19
event 6-18
transition action 6-19
transition action 2-33
trigger type
function call 3-18

tutorial
carousel
add and define data task 8-20
add and define events task 8-18
basic operation 8-2
create a Simulink model task 8-9
create and label AND states task 8-14
create and label OR states task 8-10
create and label superstates task 8-13
create default transitions task 8-17
create the transitions task 8-15
design questions 8-4
determining data 8-7
determining default transitions 8-7
determining events 8-7
determining state decomposition 8-5
determining state labels 8-6
determining states 8-4
determining transitions 8-6
label the transitions task 8-16
objectives 8-2
overview 8-2
provided models 8-2
sample Stateflow diagram 8-10
save the model task 8-23
task overview 8-8
typographical conventions xxvii

U

unary actions 6-48
unary operations 6-47
user-written functions 6-49

V

virtual scrollbar
example of adjusting font size 2-8
glossary definition A-11
overview 2-7